Razvoj web aplikacije uz primjenu Spring, Docker i Kubernetes

Lazar, David

Master's thesis / Diplomski rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:453354

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

David Lazar

Razvoj web aplikacije uz primjenu Spring, Docker i Kubernetes

DIPLOMSKI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

David Lazar

JMBAG: 0016122928

Studij: Informacijsko i programsko inženjerstvo

Razvoj web aplikacije uz primjenu Spring, Docker i Kubernetes

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Dragutin Kermek

Varaždin, rujan 2022.

David Lazar

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

U radu se opisuju osobine web aplikacije. Programski okviri za Java web aplikacije, s naglaskom na programski okvir Spring. Arhitektura modernih web aplikacija, odnosno Spring MVC arhitektura. Prednosti i mane Springa. Što su to web i RESTful servisi te njihova primjena. Što je to Docker i primjena kontejnera u web aplikacijama. Kako se izvršava automatska isporuka pomoću Kubernetes. U praktičnom dijelu rada se izrađuje baza podataka za aplikaciju te se razvija web aplikacija tako da se aplikacija u programskom okviru Spring zapakira u Docker kontejner te se isporuči aplikacija pomoću Kubernetes.

Ključne riječi: web aplikacija; Spring; Spring MVC; Docker; Kubernetes; RESTful servisi; web servisi;

Sadržaj

1. Uvod

U današnje vrijeme internet je prisutan svugdje te je najpopularniji medij za širenje informacija pa tako gotovo svaka organizacija, udruga, poduzeće ima svoju web (skraćeno od *World Wide Web*) stranicu. Zbog toga raste popularnost izrada web aplikacija. Za razvoj web aplikacije najčešće se koriste različiti programski okviri (*eng.* Framework). Postoje programski okviri za izradu korisničkog dijela (*eng.* Frontend) i klijentskog dijela (*eng.* Backend) aplikacije. Ovaj rad bazira se na programskim okvirima klijentskog dijela aplikacije koji koriste programski jezik Java. U radu će se usporediti nekoliko programskih okvira sa Springom, programskim okvirom koji je korišten u izradi aplikacije. Programski okvir Spring služi za izradu web servisa koji će dohvaćati podatke iz baze, izvršavat poslovnu logiku te će slati potrebne informacije korisničkom dijelu. Korisnički dio aplikacije izrađen je pomoću korisničkog okvira Angular koji koristi programske jezike TypeScript i HTML te će dobivene podatke sa web servisa prikazivati korisniku.

Nakon što se razvije web aplikacija, potrebno ju je postaviti na web kako bi bila dostupna široj količini ljudima. To se radi pomoću kontejnerizacije aplikacija, odnosno spremanje cjelovitog kôda i svih ovisnosti (*eng.* Dependency) aplikacije u kontejner. Jedna od prednosti kontejnerizacije je izolacija aplikacije od operacijskog sustava. Programska platforma koja je zaslužna za kontejnerizaciju je Docker. Ukoliko se aplikacija sastoji od više kontejnera potrebno ih je pravilno rasporediti, a za to služe alati za orkestraciju kontejnera od kojih je jedan od najpoznatijih Kubernetes.

Ovaj rad se sastoji od dva dijela: teoretskog i praktičnog. U teoretskom dijelu rada opisuju se elementi koji su ranije spomenuti. Dakle, prvo će biti nešto više rečeno o web aplikacijama, zatim će biti opisane sličnosti i razlike između SOAP i REST web servisa, pa će slijediti usporedba nekoliko programskih okvira sa Springom te na kraju će detaljnije biti opisani Docker i Kubernetes, odnosno programske platforme za kontejnerizaciju i orkestraciju kontejnera. Nakon toga slijedi praktički dio koji se sastoji od aplikacije *Autoškola* koja je izrađena pomoću ranije navedenih tehnologija. Glavne funkcije aplikacije su rješavanje teoretskog ispita autoškole te vođenje dnevnika vožnje od strane instruktora.

2. Metode i tehnike rada

Metode i tehnike rada koje se koriste u ovom radu su prvenstveno istraživačke, odnosno potrebno je napraviti Web aplikaciju pomoću Springa, Dockera i Kubernetesa, a zatim istražiti njihove srodne alate i usporediti ih. Tako treba istražiti alate za izradu serverskog dijela web aplikacija sličnih Springu, alate za kontejnerizaciju sličnih Dockeru te alate za orkestraciju kontejnera sličnih Kubernetesu.

Alati korišteni za izradu aplikacije su sljedeći :

- MySQL Workbrench – alat za manipuliranje bazama podataka, korišten za povezivanje na bazu podataka, manipuliranje podacima te za uvoz/izvoz podataka
- Spring Tools Suite – alat za izradu aplikacije u Springu, odnosno za izradu serverskog dijela aplikacije u programskom jeziku Java
- Angular – alat za izradu klijentskog dijela aplikacije pomoću HTML-a, CSS-a i TypeScript-a
- Docker – alat za kontejnerizaciju pomoću kojeg se kontejneriziraju serverski i klijentski dio aplikacije te baza podataka
- Kubernetes – alat za orkestraciju kontejnera

Ostali korišteni alati :

- Draw.io – alat korišten za crtanje dijagrama
- IntellliJ – alat korišten za generiranje dijagrama

3. Općenito o web aplikacijama

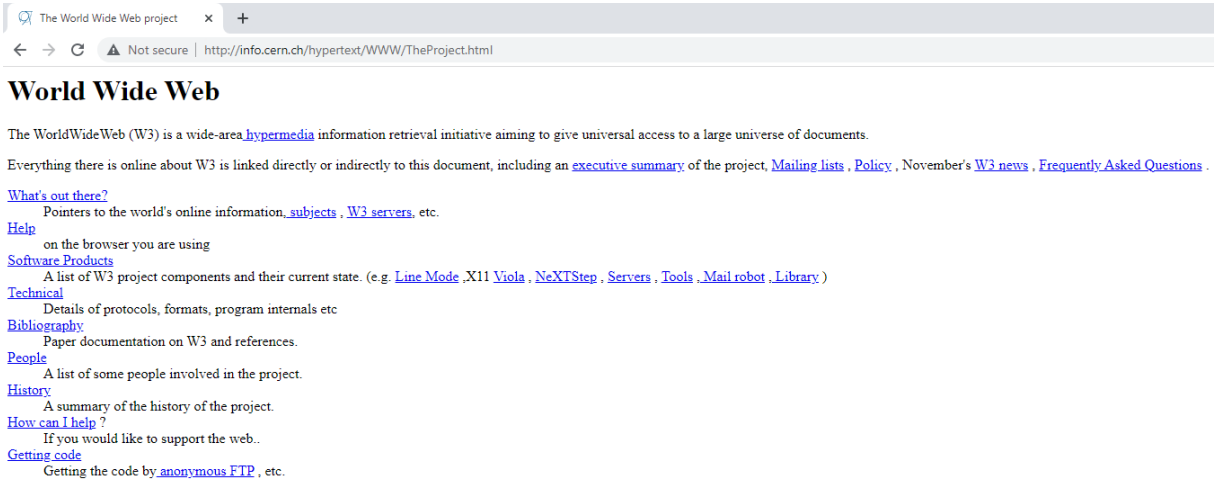
Web aplikacije su aplikacije slične aplikacijama koje se pokreću na radnoj površini, ali glavna razlika je da se web aplikacija pokreće u web pregledniku (*eng. Browser*). To znači da je Web aplikacija smještena na serveru, a pristupa joj klijent preko web preglednika (*eng. Browser*) pa se ovdje odmah može primijetiti klijent-server arhitektura, koja će više biti objašnjena nešto kasnije. Primjeri web aplikacije su poslužitelji elektroničke pošte (npr. Gmail), društvene mreže (npr. Facebook), web stranice za reproduciranje videa (npr. YouTube) itd.

Važno je razlikovati web aplikacije i izvorišne (eng. *Native*) aplikacije. Glavna razlika je u tome da izvorišne aplikacije trebaju imati različite implementacije ovisno na kojem se operacijskom sustavu koriste te se moraju na svaki uređaj posebno instalirati, dok za web aplikacije to nema nikakve veze, zbog toga što se njima pristupa preko preglednika.

3.1. Povijest web aplikacija

Kako bi postojala web aplikacija, odnosno aplikacija dostupna na internetu, potrebno je prvo izumiti internet pa tako sama povijest web aplikacija počinje pojavom interneta. Preteča današnjeg interneta prvi put se pojavljuje 1969. godine pod nazivom ARPANET koje je razvilo američko ministarstvo znanosti. Kratica ARPA označuje Agenciju za napredne istraživačke projekte (*eng. Advanced Research Project Agency*), dok NET označuje mrežu (*eng. net*). ARPANET je služio za povezivanje sveučilišta i istraživačkih centara. Sljedeći važni događaj za razvoj interneta dogodio se 1989. godine kada je Tim Bernes-Lee napisao dokument pod nazivom „*Information Management: A Proposal*“. Bernes-Lee je ovim dokumentom dao prijedlog za upravljanjem općim informacijama u CERN-u kako bi se spriječio gubitak informacija. Kako u dokumentu navodi Bernes-Lee[1], on je pokušao riješiti problem spremanjem informacija u strukturi stabla ili pomoću ključnih riječi, ali je opisao probleme zbog kojih je to bilo neučinkovito. Problem stabla je da informacije prirodno ne mogu biti spremljene u strukturu stabla jer ako list ne sadržava referencu na određenu informaciju, ponovno bi trebao ući u sustav kako bi se došlo do informacija. Problem kod spremanja informacija pomoću ključnih riječi je da dvoje ljudi nikad ne odabere iste ključne riječi i tada su ključne riječi od pomoći samo ljudima koji već poznaju aplikaciju. Tada je jedino logično rješenje bilo razviti sustav pomoću hiperteksta. Hipertekst u to vrijeme nije bio strani pojam, jer je sam Bernes-Lee 1980. razvio program Enquire koji je dopuštao spremanje informacija te pristup njima pomoću poveznica od jedne stranice do druge. Sličnu

aplikaciju je razvio Apple za Macintosh koja se zvala Hypercard. Razlika između Enquirea i Hypercarda, kako Bernes-Lee[1] navodi su sljedeće: Enquireu je nedostajalo ljepše grafike, pokretao se na više korisničkom sustavu i puno ljudi je imalo pristup do podataka. U dokumentu se također već spominje klijent-server infrastruktura koja će se više spominjati u poglavlju Arhitektura web aplikacija. Nakon objave ovog dokumenta Bernes-Lee je krenuo dalje razvijati svoju ideju te je uz pomoć kolege Roberta Cailliau-a dao ozbiljniji prijedlog zvan World Wide Web. Do 1990. Bernes-Lee i njegov tim izradili su jezik za označavanje hiperteksta (HTML), protokol za prijenos hiperteksta (HTTP), prvi web preglednik (WorldWideWeb), prvi web poslužitelj (poslije poznat kao CERN HTTPd) te prvu web stranicu koja se nalazi na slici ispod.



Slika 1. Prva web stranica (preuzeto sa http://info.cern.ch/)

Kao što je vidljivo na slici, prva web stranica sastoji se od teksta i poveznica. U tekstu se objašnjava što je to World Wide Web (W3) i kako funkcionira te sam tekst sadrži poveznice na druge web stranice, npr. poveznica „hypermedia“ vodi na stranicu gdje se objašnjava pojmovi poput hiperteksta i hipermedije.

Prvi web preglednik World Wide Web također je bio i editor, tako da se s njim mogla i uređivati web stranica. Nakon World Wide Weba, kako navode Grosskurth i Godfrey [2], gotovo u isto vrijeme istraživači Sveučilišta Kansas razvili su preglednik zvan Lynx koji je radio samo s tekstom i hipertekstom te je 1993. godine prilagođen da podržava web. Nakon Lynxa, Nacionalni Centar za Superračunalne aplikacije (NCSA) je razvio preglednik Mosaic, koji je imao grafičke značajke pa su se u njemu mogle prikazati slike između teksta. Godinu

dana poslije Bernes-Lee osnovao je World Wide Web konzorcij (W3C) kojemu je zadatak voditi razvoj weba i promovirati interoperabilnost između web tehnologija, odnosno razvijanje protokola i smjernica koji osiguravaju dugoročni napredak weba. Konzorcij se danas sastoji od 400-tinjak organizacija gdje svi imaju jednak cilj te sam konzorcij nema jedinstveno sjedište, već su četiri organizacije kao domaćini konzorciju: MIT (Amerika), ERCIM (Francuska), Keio University (Japan) i Beihang University (Kina). Nakon osnivanja konzorcija, sljedećih nekoliko godina slijedio je „rat“ web preglednika. Web preglednici koji se ističu su Netscape Navigator koji se za godinu dana razvio u Netscape2 i Microsoftov Internet Explorer (IE) 1.0, odnosno IE2 već iste godine. Godinu poslije već je izašao IE3 koji je uz kolačiće, SSL i JavaScript podržavao Java programe, ActiveX i CSS. Tako je 1998. već izašao i IE4 koji je podržavao dinamični HTML koji je pokrenuo Web 2.0 revoluciju, odnosno početak razvoja web aplikacija. Netscape nije mogao držati korak s Microsoftom pa je 1998. godine razvio Mozillu, te je 2003. prodao Mozillu, a zatim otišao u stečaj. Web preglednici koji se danas koriste uz Internet Explorer i Mozillu Firefox su Google Chrome, Opera te Safari. Kao što je prije spomenuto, IE4 pokrenuo je revoluciju s izradom web aplikacija, a dodatan razlog tome je bio što su u to vrijeme već bili razvijeni JavaScript i XML. JavaScript je jedan od najvažnijeg elementa weba uz HTML i CSS, dok je XML jezik za označavanje koji služi za pohranu i prijenos podataka koji su pomoću određenih pravila čitljivi i ljudima i računalu. Najveća revolucija u izradi web aplikacija je bila sredinom 2000-ih godina kada se pojavio AJAX (skraćeno od Asinkroni JavaScript i XML) koji je zapravo skup tehnika za razvoj weba koji se odvija na strani klijenta kako bi se razvila asinkrona web aplikacija, odnosno AJAX omogućuje slanje/dohvaćanje podataka bez preuzimanja cijele web stranice. Prva aplikacija koja je koristila AJAX bila je Gmail, a zatim su se počele razvijati druge, više društvene aplikacije kao što su YouTube, MySpace, Facebook.

3.2. Osobine web aplikacija

U prethodnom poglavlju pričalo se o razvoju weba i web aplikacija kroz povijest, a kako su se razvijale web aplikacije i tehnologije tako su se razvijale i osobine web aplikacija. Web aplikacije možemo podijeliti u tri generacije, odnosno na Web 1.0, Web 2.0 te Web 3.0 aplikacije. Svaka generacija web aplikacija ima svoje osobine, ali također i neka ograničenja.

3.2.1. Osobine Web 1.0 aplikacija

Začetnik Web 1.0 aplikacija je Tim Bernes-Lee, a njihovo razdoblje trajalo je od 1989. do 2005. godine. Web 1.0 naziva se još i Hipertekst Web te je služio samo za čitanje sadržaja. Prema Khanzodu [3] Web 1.0 koristi glavne web protokole, HTTP, HTML te URI, a njegove karakteristike su sljedeće:

- Sadržaj služi samo za čitanje
- Uspostavljanje online prisutnosti tako da je dostupno svima
- Uključuje statične web stranice koristeći osnovni HTML

Kao što se može primijetiti, karakteristike Web 1.0 aplikacija su osnove svih sljedećih web aplikacija, a to su čitljivost sadržaja, dostupnost svima te korištenje HTML jezika za prikaz sadržaja na strani klijenta. Također Khanzode [3] navodi i ograničenja za Web 1.0:

- Web stranice su razumljive samo ljudima te nemaju sadržaj kompatibilan računalu
- Postoji samo jedna osoba (web majstor) koja je odgovorna za ažuriranje korisnika i upravljanjem sadržaja web stranice
- Nedostatak dinamike, tj. niti jedna konzola nije sadržavala dinamičke događaje

Rješavanjem tih ograničenja, koja su zapravo za većinu današnjih aplikacija nedostaci, razvila se druga generacija web aplikacija, odnosno Web 2.0.

3.2.2. Osobine Web 2.0 aplikacija

Začetnici Web 2.0 aplikacija su Tim O'Reilly i Dale Doughtery koji su 2004. godine na konferenciji za Media Live International došli do koncepta za web koji nije samo za čitanje, veći i za pisanje sadržaja. Web 2.0 se još naziva i Socijalni Web. Dok je Web 1.0 imao milijune korisnika koji su mogli samo čitati na webu, Web 2.0 ima milijarde korisnika koji su mogli čitati sadržaj, a istovremeno i stvarati sadržaj.

Khanzode [3] navodi da je najbolje razumjeti osobine Web 2.0 kroz tri različita aspekta:

- Osobine usmjerene na tehnologiju – web je postao platforma sa aplikacijama koje su došle na razinu da se mogu koristiti na više od jednog uređaja. Tehnologija koja je povezana sa blogovima, podcastima, RSS izvorima i slično.
- Poslovno usmjerene osobine – način projektiranje aplikacije i poslovanja. Poslovna revolucija je u računalnoj industriji uzrokovana prelaskom na Internet kao platformu i pokušajem razumijevanja pravila na novoj platformi
- Osobine usmjerene na korisnika – Socijalni Web često se koristi za karakterizaciju stranica koje se sastoje od zajednica. Riječ je o upravljanju sadržajem i novim načinima komunikacije i interakcije među korisnicima. Web aplikacija olakšava kolektivnu proizvodnju znanja, društveno umrežavanje i povećava razmjenu informacija.

Iz navedenih osobina može se utvrditi da se Web 2.0 fokusira na korisnika, jer se čak u osobinama usmjerih na tehnologiju spominju blogovi, RSS izvori gdje su glavni akteri korisnici, isto kao i u poslovnim usmjerenim izvorima gdje poduzeće prelaze na Internet kao platformu, npr. Internet trgovina gdje se korisniku nude različiti proizvodi. Ovdje je Khanzode naveo dosta osobina, ali su Liu i suradnici [4] dodali još neke koje je Khanzode izostavio, a to su dinamički sadržaj, meta podaci, skalabilnost, ponovo iskoristiv mikro-sadržaj.

3.2.3. Osobine Web 3.0 aplikacija

Web 3.0 razvio je Berners-Lee 2016. godine. Kako je Web 1.0 bio samo za čitanje, Web 2.0 za čitanje i pisanje, Web 3.0 se najlakše može opisati da je za čitanje, pisanje i izvršavanje. Drugi naziv za Web 3.0 je Izvršni Web, ali je poznatiji kao Semantički Web. Semantički Web pruža zajednički programski okvir koji omogućuje dijeljenje i ponovnu upotrebu podataka preko aplikacija i poduzeća. U Semantičkom Webu nestaje koncept web stranice te se podaci ne posjeduju već dijele preko različitih aplikacija i servisa.

Prema Khanzodu [3] glavne karakteristike Web 3.0 aplikacije su :

- SaaS (*eng.* Software-as-a-service) poslovni model
- Softverska platforma otvorenog koda
- Distribuirana baza podataka (svjetska baza podataka)
- Personalizacija weba
- Udruživanje resursa
- Inteligentni web

Prema karakteristikama za Web 3.0 može se utvrditi da se fokusira na što jednostavnijem korištenju weba za mnoštvo korisnika. Primjerice, SaaS poslovni model pruža aplikaciju kao uslugu, gdje korisnik dobije aplikaciju, a on sam ne mora voditi računa ni o platformi, ni o spremanju podatka, zapravo ni o čemu već za njega to radi davatelj usluga. Također se danas analizira korisnik weba, sve što on radi i onda prema tome mu se nude reklame ovisno o tome što on pretražuje. To je neka posljedica karakteristika personalizacija weba i inteligentnog weba.

3.3. Arhitektura web aplikacija

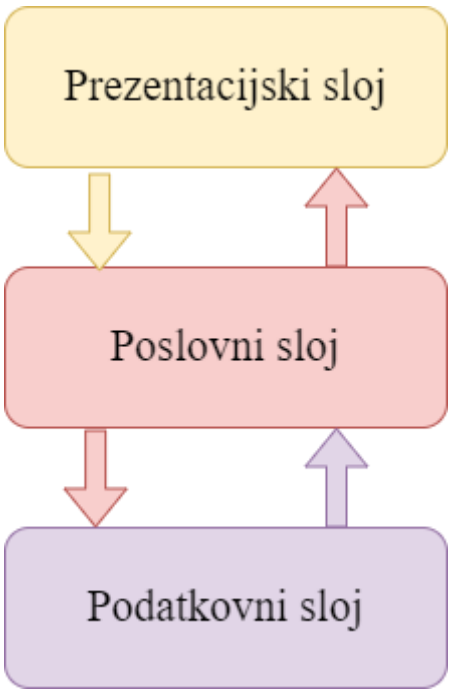
Višeslojna arhitektura web aplikacija jedna je od najpoznatijih arhitekturnih uzorka koji se koristi za izradu srednjih i velikih web aplikacija. Jedna od glavnih prednosti slojevite arhitekture jest to da se ona sastoji od više slojeva te svaki sloj ima svoj zadatak, neovisan je o drugim slojevima te komunicira samo sa susjednim slojevima. Višeslojna arhitektura može imati *n* slojeva. Tako se aplikacija s jednim slojem naziva jednoslojna ili jednostavna aplikacija, aplikacija s dva sloja dvoslojna itd.

Jednoslojne aplikacije najjednostavnije su aplikacije gdje se sve odvija u jednom sloju. Jedan od primjera jednoslojne aplikacije je aplikacija izrađena samo u Angularu. U tom jednom sloju se nalazi sva poslovna logika, spremaju se podaci te se prikazuju korisniku. Takve aplikacije najčešće se rade kod učenja nekog programskog jezika te nemaju neku veliku važnost.

Dvoslojne aplikacije su nešto složenije. One imaju dva sloja, a to su klijent i poslužitelj. Dvoslojne aplikacije se mogu podijeliti na dva modela, a to su klijent-poslužitelj s bogatim klijentom i klijent-poslužitelj s bogatim poslužiteljem. Razlika između ova dva modela je gdje se vrši većina poslovne logike pa se tako u klijent-poslužitelj s bogatim klijentom modelu

većina poslovne logike vrši na strani klijenta, dok se u drugom modelu većina poslovne logike vrši na poslužitelju.

Troslojna arhitektura sastoji se od tri sloja, a to su podatkovni sloj, poslovni sloj te prezentacijski sloj (Slika 2.). Kao što je prije navedeno, svaki sloj stoji za sebe te može komunicirati samo sa susjednim slojem. Ovi slojevi su podijeljeni predstavljaju i fizički i logički odvojene cjeline.



Slika 2. Prikaz troslojne arhitekture, prema [5]

Podatkovni sloj (*eng.* Data Presistance Layer), bavi se isključivo upravljanjem podataka, odnosno upravlja CRUD (*eng.* Create Read Update and Delete) operacijama nad bazom podataka. Grgić navodi [5] da podatkovni sloj može upravljati jednim ili više spremištem podataka te se prema zahtjevima klijenta i aplikacije definiraju ta spremišta. Podaci o konfiguraciji baze podataka spremaju se u poseban dokument kojem pristup ima jedino podatkovni sloj. Poslovni sloj vidi podatkovni sloj u obliku crne kutije sa sučeljima za pristup podacima, odnosno poslovni sloj ne vidi ništa što se događa u podatkovnom sloju, samo zna metode koje ima podatkovni sloj.

Poslovni sloj (*eng.* Business Logic Layer) sadržava metode koje opisuju cijelu poslovnu logiku te koristi podatkovni sloj kako bi dohvatio potrebne podatke, vršio operacije nad njima te spremio promjene. Prema Grgić [5] unutar poslovnog sloja nalaze se objekti koji opisuju poslovna pravila, poslovna logika koja izražava sve politike i zahtjeve korisnika, tijekovi rada

koji definiraju način prenošenja podataka iz jednog modula u drugi. Jedan od načina za postizanje sigurnosti na poslovnoj sloju je uvođenje uloga, kako bi pristup podacima imale samo ovlaštene osobe.

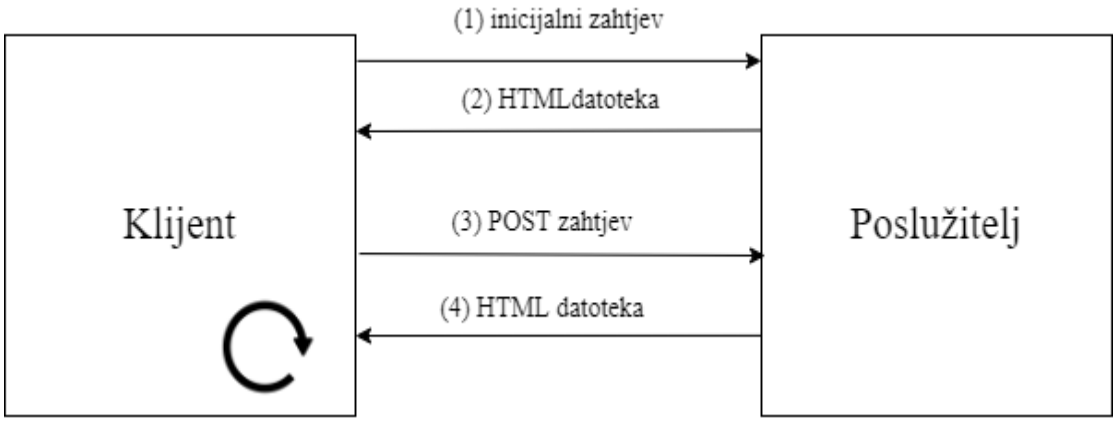
Prezentacijski sloj (*eng.* Presentation Layer) ili poznat kao korisničko sučelje može se nalaziti na više platformi, primjerice na mobilnom uređaju, tabletu ili webu. Uloga prezentacijskog sloja je prikaz podataka korisniku te interakcija između korisnika i aplikacije, odnosno kroz forme prikazati rezultat upita korisniku.

3.4. Vrste web aplikacija

Web aplikacije se može podijeliti na više različitih načina, tako se mogu podijeliti prema platformama na kojima se prikazuju na web i mobilne aplikacije. Mobilne web aplikacije se još mogu podijeliti ovisno za koji operacijsku sustav su pisane na iOS ili Android aplikacije, no ovdje će se podijeliti ovisno o načinu učitavanja stranica pa se tu dijele na web aplikacije na jednoj stranici (*eng.* Single Page Application), gdje se učita samo jedna stranica na kojoj se samo mijenja sadržaj i na web aplikacije na više stranica (*eng.* Multiple Page Application), gdje se svaka stranica mora posebno.

3.4.1. Web aplikacije na više stranica (MPA)

Web aplikacije na više stranica (*eng.* Multiple Page Application) ili tradicionalne web aplikacije sastoje se od nekoliko stranica sa sadržajem (tekst, slika, video, i sl.) koje sadrže poveznice na druge stranice. Prilikom prelaska s jedne stranice na drugu, web preglednik mora ponovo učitati cijeli sadržaj web stranice te ponovno preuzeti potrebne podatke za prikaz te stranice, čak i dijelove stranice koji su jednaki, npr. zaglavlje i podnožje.



Slika 3. Životni ciklus aplikacije na više stranica, prema [6]

Na slici 3. prikazan je životni ciklus web aplikacije na više stranica. Dakle, prvo kada korisnik pristupi web stranici prvo šalje inicijalni zahtjev na poslužitelj (1). Nakon što poslužitelj obradi zahtjev, šalje klijentu na preglednik HTML datoteku sa osnovnim informacijama (2). Tada klijent pošalje zahtjev za dohvat podataka na drugoj stranici na poslužitelj kroz navigaciju (3) te poslužitelj opet obradi zahtjev te pošalje novu HTML datoteku klijentu (4).

Web aplikacija na više stranica nije loša za aplikacije koje sadrže nekoliko stranica sa statičnim sadržajem, no problem dolazi kada korisnik traži bogato korisničko sučelje gdje je potrebno puno osvježavanje podataka, jer bi se svakim osvježavanjem podataka trebala učitati nova stranica. Pojavom AJAX-a sredinom 2000-tih godina riješio se taj problem, pa se stranica mogla osvježiti bez preuzimanja cijele stranice, tako da su se mogli preuzeti i osvježiti samo potrebni podaci. Prema Soloveu [6] neke od prednosti web aplikacija na više stranica su :

- Pruža bolju vizualizaciju aplikacije, višerazinska navigacija
- Popularan način za izradu web aplikacije jer ima puno literature

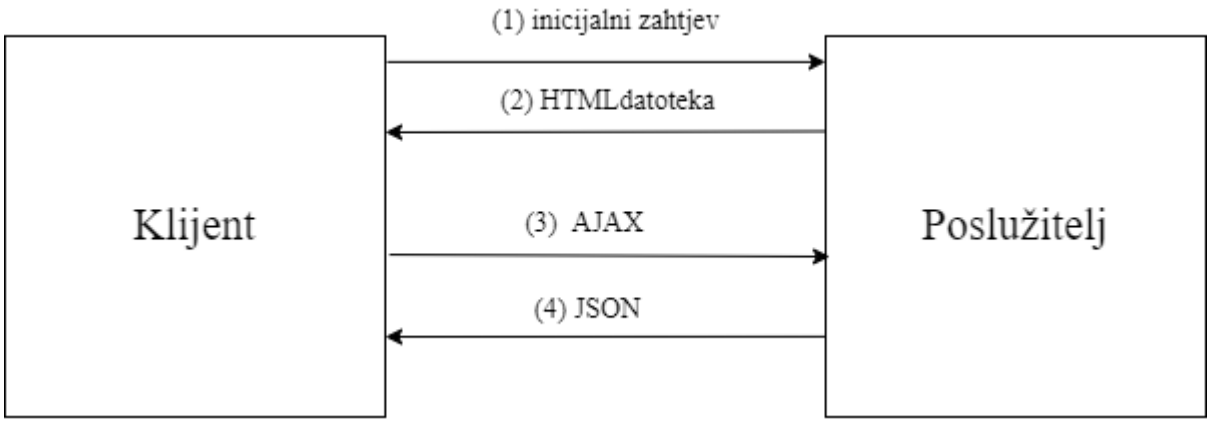
Nedostaci web aplikacija na više stranica prema Soloveu [6] :

- Ne dopušta promjenu poslužiteljskog dijela bez utjecaja na klijenta
- Razvoj postaje kompleksniji, potrebno je koristiti programske okvire bilo za klijentski ili poslužiteljski dio aplikacije što produžuje vrijeme razvoja
- Ako se ne koristi AJAX, cijela stranica će se ponovo učitavati nakon svakog poziva

- Razvoj mobilnih aplikacije traje puno dulje jer treba poslužiteljski dio raditi od početka

3.4.2. Web aplikacije na jednoj stranici (SPA)

Web aplikacije na jednoj stranici (*eng.* Single Page Application) su doslovno web aplikacije koje sadrže samo jednu HTML datoteku učitanu u pregledniku i ona se nikada ponovo ne učitava. Svaki pozivom one učitavaju samo potrebne podatke, a ne cijelu HTML datoteku. To je zbog toga što svaka web aplikacija na jednoj stranici sadrži AJAX koji vraća JSON podatke umjesto HTML datoteke. Moguća su dva načina implementacije takve aplikacije. Jedna je da se odmah preuzme sav potreban kôd (CSS, HTML, JavaScript) na klijentski dio aplikacije, a drugi način je da se dinamički preuzme potreban kôd prilikom odgovora na korisničke operacije.



Slika 4. Životni ciklus aplikacije na jednoj stranici, prema [6]

Na slici 4 je prikazan životni ciklus aplikacije na jednoj stranici. Početak životnog ciklusa je jednak kao i kod aplikacije na više stranica, ali ovdje jednom kada se učita HTML datoteka, više se ona ne preuzima, nego klijet prema poslužitelju šalje AJAX zahtjev dok poslužitelj obrađuje zahtjev i vraća JSON odgovor sa potrebnim podacima.

Web aplikacije na jednoj stranici rade samo na klijentskom dijelu te kako nema ponovnog učitavanja stranica, korisnik može koristiti aplikaciju bez problema. Neki od primjera takvih aplikacija su Gmail, Facebook, Google Maps. Danas ima dosta programskih okvira koji pomažu razvijati web aplikaciju na jednoj stranici, a najpopularniji među njima su React, Angular, VueJS, ASP.NET. Kod takvih aplikacija korisnik se može „šetati“ po aplikaciji kada prolazi kroz navigaciju što je jako pristupačno korisniku. Prema Soloveu [6] prednosti web aplikacije na jednoj stranici su:

- Rade na više uređaja
- Jednostavniji razvoj aplikacije, nije potrebno napisati kod za prikazivanje na poslužitelju
- Postoje proširenja na preglednicima za lakše otklanjanja poteškoća
- Može učinkovito spremiti bilo koju lokalnu pohranu u predmemoriju
- Lakše je izgraditi bogatije korisničko sučelje jer se radi samo na jednoj stranici
- Dugoročno niske cijene za održavanje

Nedostaci web aplikacija na više stranica prem Soloveu [6] :

- Teški klijentski programski okviri koji se moraju učitati klijentu
- Neke web tražilice ne podržavaju prikazivanje stranica na strani klijenta, što ne dopušta implementaciju učinkovitog upravljanja
- Dupliciranje usmjerivača (*eng.* Router) (u usporedbi s klasičnim pristupom)
- Budući da ima jedan ulaz u aplikaciju, postoji rizik da jedna greška može dovesti do neoperabilnog stanja cijele aplikacije

4. Servisi web aplikacija

Kako bi se bolje razumjelo o čemu će se pisat u ovom poglavlju, najbolje je prvo definirati što su to web servisi. Definicija o web servisima ima dosta, ali ovdje će se usporediti dvije definicije.

Tako su prema Yangu [7] web servisi samostalne aplikacije dostupne na webu koje su sposobne ne samo obavljati poslovne aktivnosti, već također posjeduju mogućnost uključivanja drugih web servisa kako bi se dovršile poslovne transakcije višeg reda.

Prema Yangovoj definiciji web servisi su aplikacija koja stoji sama za sebe te su dostupne na internetu, što znači da im se može pristupiti ako se ima pristup internetu. Osim što je Yang definirao što su web servisi, u svojoj definiciji dodaje što oni rade pa tako prema Yangu web servisi osim što obavljaju poslovnu logiku za koju su zaduženi, također imaju mogućnost pozivati druge web servise kako bi se mogle izvršiti kompliciranija poslovna logika. Kao što je u prvom poglavlju navedeno da se konzorcij (W3C) bavi razvojem web i njegovih tehnologija, oni su također definirali web servis.

„Web servis je programski sustav osmišljen za podršku interoperabilne interakcije stroj-stroj preko mreže. Ima sučelje opisano u formatu koji se može strojno obraditi (točnije WSLD). Drugi sustavi stupaju u interakciju s web servisom na način propisan opisom koristeći SOAP poruke, koje se obično prenose pomoću HTTP-a s XML serijalizacijom u kombinaciji s drugim standardima povezanim s webom.“ [8]

Kada se usporede Yangova i definicija od konzorcija, mogu se vidjeti neke sličnosti, ali i razlike. Sličnost definicija je da su web servisi dostupni na webu te da jedan web servis može koristiti drugi. Razlika je u tome da Yang spominje što web servis radi (obavlja poslovne aktivnosti) dok definicija konzorcija više govori o načinu kako se izvršava komunikacija sa web servisom i koje tehnologije su potrebne za to.

4.1. Vrste web servisa

Potreba za web servisom počela je kada je došlo do potrebe za komunikacijom između procesa. Takva komunikacija odvija se na odvojenim računalima povezanim lokalnom mrežom (*eng.* Local Area Network) pomoću mrežnih utičnica (*eng.* Socket). Sljedeći problem bio je da komunikacija između procesa na odvojenim računalima nije radila ako su arhitekture sustava drugačije, zbog toga što je svaka arhitektura imala različitu tehniku predstavljanja podataka. Kako bi se riješio ovaj problem razvio se je XDR (*eng.* External Data Representation) format koji definira zajedničke standarde za prikaz podataka.

Nakon pojave interneta i weba, XDR je evoluirao u XML. Priroda rada im je jednaka, ali razlika je bila u tome što je XDR binarnog dok je XML tekstualnog formata. Uz pojavu XML-a razvio se SOAP (*eng.* Simple Object Access Protocol) protokol, a kasnije i REST (*eng.* REpresentational State Transfer) web servisi koji su dvije najvažnije vrste web servisa koje se danas koriste, a detaljnije o njima biti će opisano dalje u tekstu.

4.1.1. SOAP web servis

SOAP je lagan i neovisan protokol za razmjenu poruka koji omogućuje komunikaciju između aplikacija pomoću HTTP-a i XML-a. Lagan je i neovisan zbog toga što nije važno s kojeg operacijskog sustava se usluga koristi. Budući da se radi o protokolu on ne pamti prošla spajanja i komunikaciju pa se zbog toga svaka aplikacija prilikom slanja zahtjeva mora predstaviti drugim aplikacijama. Prema Mariću [9], arhitektura SOAP web servisa sastoji se od tri dijela, a to su jezik za opisivanje web servisa koji predstavlja standard za opisivanje mogućnosti web servisa, UDDI standarda za objavljivanje web servisa te SOAP protokola za komunikaciju između web servisa.

Jezik za opisivanje web servisa (*eng.* Web Service Description Language; dalje u tekstu WSDL) je jezik kojim se opisuje ponašanje i radnje web servisa prema SOAP poruci i obratno. WSDL dokument koji je opisan pomoću XML formata sadrži opis svih metoda koje sadrži određeni web servis i odgovora koji se mogu očekivati od svake završne točke servisa. Drugim riječima, WSDL definira sučelje pomoću kojeg druge aplikacije i servisi mogu znati koje tipove podataka prima koja metoda, te koji tip podataka vraća. WSDL je većinom namijenjen računalima za automatsko generiranje programskog koda iz WSDL dokumenta.

UDDI (*engl.* Universal Description, Discovery, and Integration) standard baziran je na XML-u koji služi za opisivanje, objavljivanje i nalaženje web servisa. Sastoji se od dva dijela: od registra svih metapodataka web servisa, uključujući pokazivač na WSDL opis servisa i od skupa definicija WSDL priključka za manipuliranje i pretraživanja tog registra. Drugim riječima, UDDI je zapravo repozitorij koji sadrži opise web servisa sadržanih u WSDL datotekama.

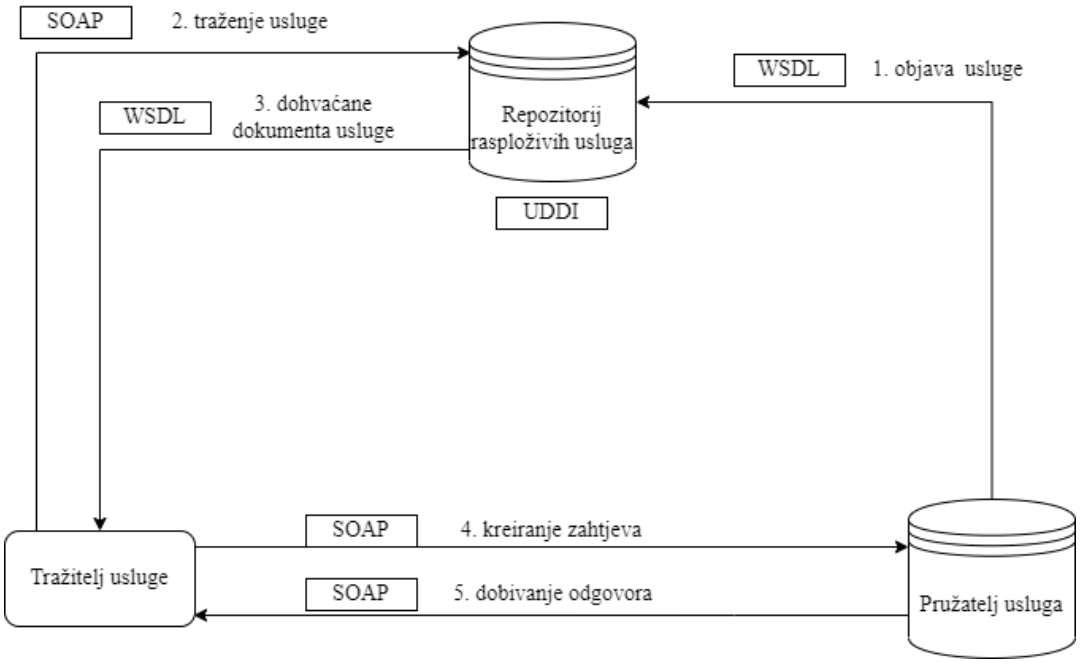
Prema Mariću [9], SOAP protokol definira format poruke koji se koristi za komunikaciju između klijenta i servisa te skup pravila kojem će se iz SOAP poruke prepoznati i pozvati metoda te kako će se rezultat obrade te metode vratiti u SOAP poruku i poslati ju natrag klijentu. SOAP poruke šalju se putem HTTP ili SMTP protokola, ali najčešće putem HTTP protokola zbog utjecajne prisutnosti u globalnoj mreži za razmjenu informacija i konfiguriranih vatrozida koji uobičajeno propuštaju promet HTTP protokola. SOAP poruke su strukturirane u SOAP omotnicu (*eng.* Enelope) čija je struktura prikazana na slici ispod. Omotnica se sastoji od dva dijela, a to su zaglavlje i tijelo.

Slika 5. Grafički prikaz SAOP omotnice, prema [10]

Prema Balentoviću [10], zaglavlje nije nužno za samo omotnicu, ali može sadržavati bitne podatke o načinu slanja i primanja, obrade, sigurnosne komponente, podatke o

transakcijama, informacije o kvaliteti usluge i drugima bez da se narušavaju specifikacije protokola. Kao što je prikazano na slici, zaglavlje dolazi uvijek prije tijela SOAP poruke. Kako je omotnica opisana XML jezikom, svaki blok unutar zaglavlja poruke naziva se zaglavni blok te zaglavlje može sadržavati više takvih blokova. Glavni dio SOAP omotnice je tijelo koje je obavezan dio. Tijelo najčešće sadrži poruke o pogrešci ili podatke kojima se koriste aplikacije. U tijelu se pozivaju metode pisane u određenom programskom jeziku, a tim metodama se pridružuju vrijednosti za ispravno izvršavanje. U tijelu se također može naći blok za pogreške koje korisnika ili aplikaciju obavještava o pogrešci i mjestu gdje se dogodila. Marić [9] navodi da se SOAP omotnica uobičajeno stavlja unutar tijela HTTP protokola koji je odvojen od svojega zaglavlja jednim praznim retkom, a to znači da se SOAP protokol koristi na sloju iznad aplikacijskog sloja.

Grafički prikaz i način rada SOAP arhitekture prikazan je na slici ispod. Dakle, glavni akteri SOAP arhitekture su tražitelj usluge, pružatelj usluge te repozitorij raspoloživih usluga. Prvi korak je da pružatelj usluga pomoću WSDL datoteke objavi svoje usluge repozitoriju raspoloživih usluga koji koristi UDDI standard za opisivanje i objavu usluga koje koristi taj web servis. Zatim tražitelj usluge preko SOAP poruke traži uslugu u repozitoriju, a repozitorij njemu šalje WSDL datoteku usluge te na kraju tražitelj usluge zna kakav SOAP zahtjev treba poslati, pa šalje zahtjev prema pružatelju usluge te on njemu vraća odgovor putem SOAP odgovora.



Slika 6. Način rada SOAP arhitekture, prema Mariću [9]

Marić [9] navodi da je glavna prednost SOAP web servisa povećana sigurnost pomoću Web Service-Securitya (WS-Secutiry). WS-Secutiry omogućava SOAP porukama povjerljivost, integritet, autentifikaciju korisnika te zaštitu od virusa. Također u SOAP web servisu pouzdanost je veća, a vjerojatnost pogreške je manja. Glavni nedostatak SOAP web servisa je da zahtjeva više mrežnog prometa od REST servisa te mu je potrebno više vremena za obradu zahtjeva i generiranje odgovora.

4.1.2. REST web servis

Kao što je ranije navedeno, REST je kratica od engleskog naziva Representational State Transfer što u prijevodu znači prijenos reprezentacijskog stanja. Prema Walker [11], REST web servis je lagani, održivi i skalabilni servis koji je izgrađen REST arhitekturom. Glavna je razlika između SOAP-a i REST-a što je SOAP protokol, dok s druge strane REST je arhitekturni stil pomoću kojeg se podaci šalju. Za slanje REST poziva i odgovora, odgovoran je HTTP protokol.

REST predstavlja način pristupa resursima koji se nalaze u nekom određenom okruženju. Ako klijent hoće pristupiti tim resursima (npr. slike, video ili drugi podaci) on mora poslati zahtjev servisu za dohvat resursa. Da bi klijent mogao pristupiti servisu, REST servis izlaže krajnje točke za dohvat resursa. Prema Walker [11], glavni elementi implementacije REST servisa su sljedeći:

- Resursi – glavni element REST servisa su resursi, odnosno URI (*eng.* Uniform Resource Identifier). URI, jedinstveni identifikator resursa, koristi se za jednoznačno definiranje resursa unutar cijele mreže. Primjerice http://autoskola.hr/ je URL do REST web servisa. Ukoliko se želi pristupiti svim korisnicima tog web servisa treba pristupiti putanji http://autoskola.hr/getUsers. Pristupom toj putanji klijent kaže web servisu da mu dohvati podatke o korisnicima zadane aplikacije.
- Glagoli zahtjeva – glagoli zahtjeva se odnose na metode koje se šalju. Najčešće četiri metode koje se šalju su: GET, POST, PUT i DELETE. Na primjer, kada se preko web preglednika pristupa poveznici koja je navedena u gornjem primjeru, preglednik šalje GET zahtjev prema web servisu.
- Zaglavlje zahtjeva – u zaglavlju zahtjeva definiraju se dodatne instrukcije. Tako se u zaglavlju zahtjeva mogu nalaziti informacije o tipu odgovora koji očekuje zahtjev, ali isto tako može sadržavati informacije o autorizaciji.

- Tijelo zahtjeva – tijelo zahtjeva sadržava podatke poslane s zahtjevom. Tijelo najčešće sadrži zahtjev koji se šalje POST metodom, dok recimo zahtjev koji se šalje GET metodom gotovo nikada ne sadržava tijelo.
- Tijelo odgovora – sadrži podatke koje servis vraća kao odgovor na zahtjev, odnosno vraćajući se na prvi primjer, tijelo odgovora na pristup putanji http://autoskola.hr/getUsers će sadržavati listu korisnika.
- Status odgovora – status odgovora se vraća zajedno sa odgovorom od web servera, a on daje informaciju kakav je status odgovora. Ako je cijela komunikacija prošla dobro, status odgovora sa servera će biti 200. Također može sadržavat i druge statusne kodove za pogreške, a sva pogreška ima svoj statusni kod.

Kao što je navedeno u glagolima zahtjeva, postoje četiri glavne metode koje se koriste: GET, POST, PUT i DELETE metoda. Prvenstveno, ove metode se razlikuju po namjeni te prema ti namjeni bi se trebale i koristiti. Namjena metoda:

- GET – koristi se za dohvaćanje resursa
- POST – koristi se za stvaranje novih resursa, npr. kreiranje novog korisnika
- PUT – koristi se za ažuriranje postojećih resursa, npr. ažuriranje podataka o postojećem korisniku
- DELETE – koristi se za brisanje postojećih resursa

Walker [11] navodi karakteristike REST servisa koji su ujedno i glavni dizajnerski principi REST web servisa, a karakteristike su sljedeće:

Klijent-server arhitektura

Najosnovniji zahtjev za arhitekturu sustava koji koristi REST servis. To znači da poslužitelj sadrži REST web servis koji klijentu treba pružiti potrebnu funkcionalnost. Klijent šalje zahtjev web servisu na poslužitelj, a poslužitelj ili odbija zahtjev ili šalje adekvatan odgovor klijentu.

Bez stanja

Koncept bez stanja znači da klijent ima odgovornost dostaviti sve potrebne informacije servisu. Bez potrebnih informacija servis ne može obraditi zahtjev klijenta. Servis ne bi trebao održavati nikakve informacije između zahtjeva klijenta, nego je to niz pitanja i

odgovora. Odnosno, klijent postavi pitanje, a servis odgovara bez da zapamti pitanja koja su prije postavljena.

Predmemorija

Koncept predmemorije treba pomoći oko problema u konceptu bez stanja. Budući da je svaki zahtjev klijenta neovisan te tako klijent može poslati više istih zahtjeva. Koncept predmemorije pamti zahtjev klijenta i ukoliko se ponovi isti zahtjev servis umjesto obrade podataka odlazi u predmemoriju i pronalazi tražene informacije. Time se štedi količina dolaznog i odlaznog mrežnog prometa od klijenta do poslužitelja i brzina odgovora servisa.

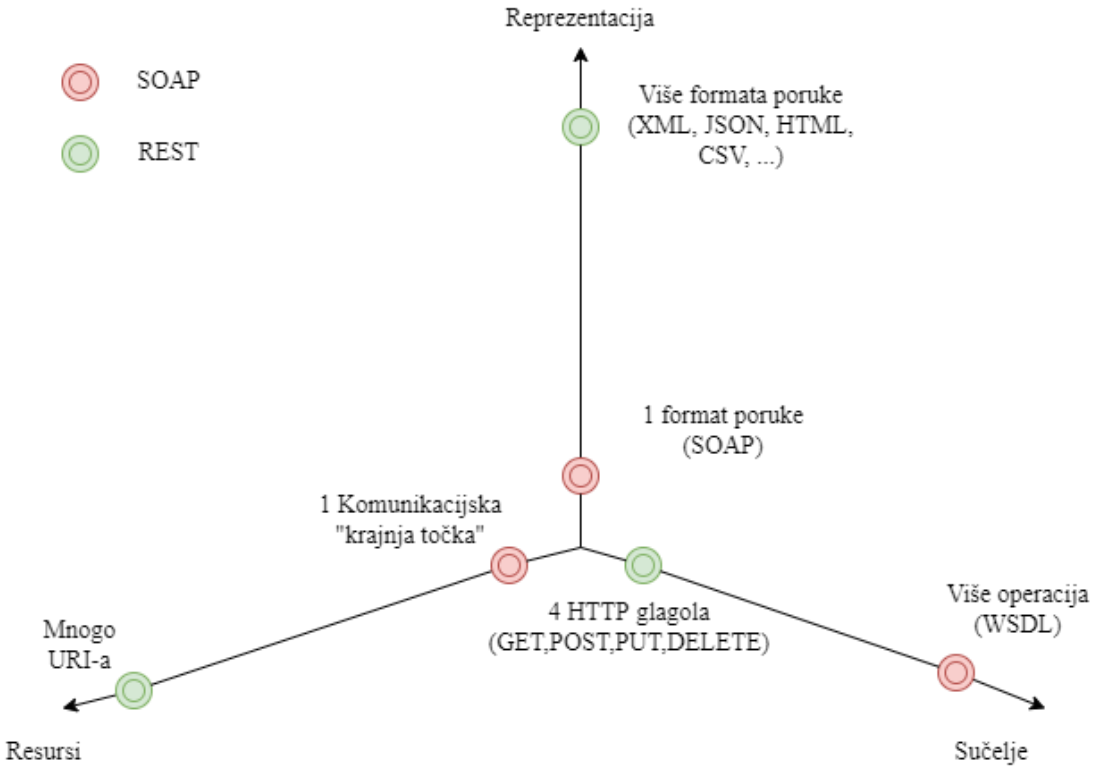
Slojeviti sustav

Koncept slojevitog sustava je da se može umetnuti dodatan sloj između klijenta i poslužitelja, odnosno da se umetne sloj gdje bi se odvijala sva potrebna poslovna logika. Taj sloj može biti neki drugi web servis, ali i ne mora. Bitno je da se taj sloj uvede transparentno kako ne bi ometao interakciju između klijenta i poslužitelja.

4.2. Usporedba SOAP i REST web servisa

SOAP i REST web servisi imaju svoje sličnosti, ali i razlike. Usporedba SOAP i REST servisa će se prikazati pomoću slika i tablica.

Pautasso je usporedio SOAP i REST servis prema tri različite kategorije, a to su resursi, reprezentacija i sučelje. U smjeru resursa može se primijetiti da SOAP web servis ima jednu krajnju točku preko koje joj se može pristupiti dok REST servis ima puno jedinstvenih identifikatora resursa. Prednost je kod SOAP web servisa da je dovoljno znati kao pristupiti jednoj krajnjoj točki, dok je prednost kod REST servisa da se može na neke URI-je dozvoliti pristup svim klijentima, dok na druge samo nekima. Što se tiče reprezentacije podataka, SOAP web servis podržava samo SOAP format poruke, dok REST servis može podržavati više formata, od kojih je najzastupljeniji JSON format. Kod sučelja može se primijetiti da REST koristi četiri HTTP metode (GET, POST, PUT, DELETE) dok su sučelja kod SOAP-a definirana pomoću WSDL datoteke te svaka metoda ima svoje sučelje.



Slika 7. Razlika između SOAP i REST servisa, prema Pautassu [12]

Balentović je napravio usporedbu tih servisa pomoću tablice gdje je naveo glavne karakteristike svakog web servisa. Prema tablici ispod, SOAP web servisi u odnosu na REST servise trebaju manji broj linija koda za implementaciju, imaju manji broj pogrešaka, pouzdaniji su, neovisni su o protokolu, sigurniji je prijenos podataka te se može izvršiti sigurna razmjena povjerljivih poruka. S druge strane REST web servisi su jeftiniji za održavanje i implementaciju, brže se izvršavaju, odnosno imaju manje mrežnog prometa, bolje performanse te mogu imati više jezika kojima su oblikovani podaci (JSON, XML). Glavna razlika je da je SOAP protokol, dok je REST stil kojim je opisana arhitektura web servisa.

Tablica 1. Usporedba SOAP i REST servisa, prema Balentoviću [10]

Kategorija	SOAP	REST
Potreban broj linija koda za implementaciju	Manji broj linija koda za implementaciju	Veći broj linija koda za implementaciju
Cijena održavanja i implementacije	Veća	Manja
Brzina izvršavanja	Manja	Veća
Količine pogrešaka	Manja	Veća
Pouzdanost	Veća	Manja
Performanse	Manje	Veće
Neovisnost o protokolu	Neovisan o protokolu	Ovisi o HTTP protokolu
Protokol preko kojega se može koristiti	HTTP, TCP, UDP, SMTP	HTTP
Jezik kojim su podaci oblikovani	XML	JSON, XML
Specifične karakteristike	-Siguran prijenos podataka -Sigurna razmjena povjerljivih poruka -Tajnost -Kredibilitet -Skalabilnost -Interoperabilnost -Prilagodljivost -Protokol	-Velike brzine obrade podataka - Laka implementacija -Lako održavanje -Korištenje URI-ja za označavanje podataka -Vrlo čitljiv -Lako ispravljanje poruka -Sve veća uporaba -Stil kojim je arhitektura opisana

Ukratko, SOAP web servisi su sigurniji, imaju manji broj pogrešaka, pouzdani su, ali danas se puno više koriste REST web servisi iz jednostavnog razloga, a to je jeftinija implementacija i održavanje, lakša i brža implementacija. Danas potreba za web aplikacijama raste pa tako i za web servisima. Želja svakog klijenta je da dobije svoj proizvod što brže i po nižoj cijeni pa to objašnjuje popularnost REST web servisa.

5. Programski okviri za Java web aplikacije

Java je visokorazinski, objektno orijentiran programski jezik kojeg je razvio James Gosling 1995. godine u Sun Microsystems. Pisan je po principu *piši jednom, pokreći svugdje* (eng. *Write once, run anywhere; WORA*) što označava da se Java kôd može pokretati na bilo kojem operacijskom sustavu. To omogućava Java virtualni stroj (eng. *Java Virutal Machine; JVM*) koji može kompilirati kôd neovisno na kojem operacijskom sustavu se pokreće. Java se je razvila iz C i C++ programskih jezika pa su ti svi jezici dosta slični po ključnim riječima te nekim značajkama. Danas tvrtka Oracle je vlasnik Jave te je Java jedan od najpopularnijih programskih jezika za razvoj poslužiteljske strane u web razvoju.

Kako se razvijala Java, tako su se razvijali programski okviri za Javu. Programski okviri pomažu programerima da što lakše i brže razviju aplikaciju. Ima puno Java web programskih okvira, ali u ovom radu biti će više opisani: Grails, Vaadin, Play te Spring.

5.1. Grails

Grails je programski okvir za izradu web aplikacija koji koristi programski jezik Groovy. Groovy je dinamički programski jezik koji se vrti na Java virtualnom stroju. Sintaksa tog programskog jezika bazirana je na Javi, dok su značajke programskog jezika inspirirane jezicima poput Ruby, Perl, Python ili Smalltalk. Fagerland i drugi [13] navode da u usporedbi s Javom, Groovy omogućuje kompaktni kôd kroz značajke kao što su metode automatskog dobivanja (*eng.* Getter) i postavljanja (*eng.* Setter) za sva javna polja, proširen API te zatvaranja (*eng.* Closure). Zatvaranje je kada funkcija može zapamtiti i pristupiti svom leksičkom opsegu čak i kada se ta funkcija izvršava izvan svog leksičkog opsega. Groovy također omogućuje integraciju s Java bibliotekama i može se jednostavno implementirati u postojeća Java okruženja.

Dizajneri Grails programskog okvira uzeli sve najbolje alate, mehanizme i tehnologije koje postoje u drugim programskim okvirima i sve to spojili u jedan napredan programski okvir. Tako je Grails baziran na Springu i Hibernateu. Sloj kontrolera je baziran prema Spring Bootu, dok se Hibernate koristi za mapiranje objekta za SQL baze podataka. Grails također koristi sustav za izgradnju temeljen na Gradleu, te ugrađeni Tomcat kontejner koji je konfiguriran za ponovo učitavanje u hodu. Grails je programski okvir ne samo za klijentski

već i za korisnički dio aplikacije. On omogućuje programerima izradu REST servisa ili modernih web aplikacija s JavaScript sučeljem.

Razvojna okruženjima (*eng.* Integrated Development Environment; IDE) u kojima se može razvijati web aplikacija pomoću Grailsa su: Netbeans IDE, IntelliJ IDEA te Eclipse/Groovy Grails Tool Suite/Spring Tool Suite. Groovy Grails Tool Suite i Spring Tool Suite su u principu nadograđeni Eclipse kako bi se prilagodili potrebnim programskim okvirima. Grails se je bolje koristiti u razvojnom okruženju Groovy Grails Tool Suite jer je više prilagođen tom okviru, ali se isto tako može koristiti u Spring Tool Suiteu, jer je sam Grails baziran na razvojnom okviru Spring.

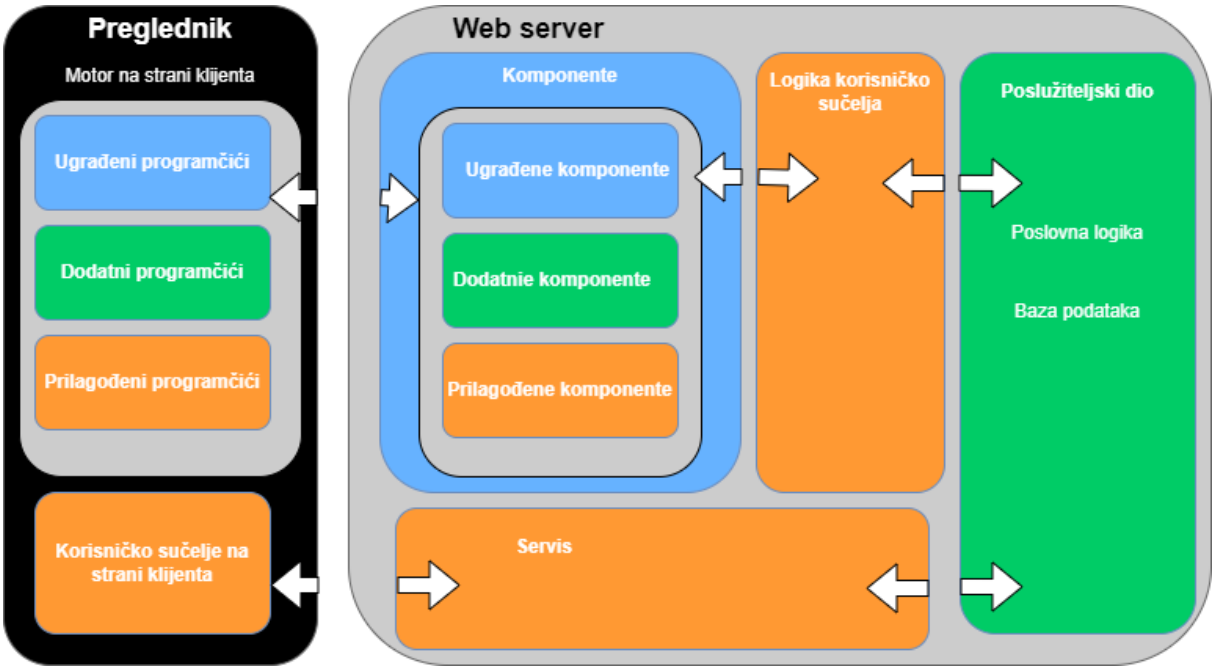
Karakteristike Grailsa su:

- Koristi programski jezik Groovy koji je dizajniran da poboljšava produktivnost razvojnih programera
- Temeljen na Spingu te koristi značajke Springa za štednju vremena, kao što je Spring ubrizgavanje ovisnosti (*eng.* Dependency Injection)
- Grails okvir primjenjuje načelo „*Nemoj se ponavljati*“ (*eng.* Don't Repeat Yourself; DRY), čime se eliminiraju ponavljanja i skrivene pogreške te omogućuju brža i lakša poboljšanja

5.2. Vaadin

Vaadin programski je okvir otvorenog kôda koji služi za izradu web aplikacija. Prema Grönroosu [14], Vaadin je dizajniran da omogući lako stvaranje i održavanje visoko kvalitetnog web korisničkog sučelja. Vaadin podržava dva različita programska pristupa: poslužiteljsku stranu i klijentsku stranu. Vaadin se više bavi poslužiteljskom stranom pa tako ona brine o upravljanju korisničkog sučelja u pregledniku i AJAX komunikacijom između preglednika i poslužitelja. Uz Vaadin pristup ne treba učiti i raditi direktno s tehnologijama za preglednik, primjerice HTMLom ili JavaScriptom.

Vaadin također koristi programski jezik Java pa razvojna okruženja u kojima se može razvijati aplikacija pomoću programskog okvira Vaadin su Eclipse IDE, InzelliJ IDEA i NetBeans IDE. Kod Eclipsa i NetBensa se treba dodatno uključiti (*eng.* Plug-in) dodatak za Vaadin, dok je kod IntelliJa već uključen.



Slika 8. Arhitektura aplikacije razvijene pomoću Vaadin okvira, prema Grönroosu [14]

Na slici 8 prikazana je arhitektura aplikacije koja je razvijena pomoću programskog okvira Vaadin. Kako je opisao Grönroos [14], arhitektura poslužiteljske aplikacije se sastoji od poslužiteljskog okvira i motora na strani klijenta (eng. *Client-side Engine).* Motor se izvršava u pregledniku kao JavaScript kôd, prikazujući korisničko sučelje i prenosi interakciju korisnika do poslužitelja. Komunikacija između klijentskog i poslužiteljskog dijela izvršava se pomoću AJAX-a, što omogućuje interaktivnost web aplikacija kakva postoji kod aplikacija koje se izvršavaju na radnoj površini. Uz razvoj Java aplikacija na strani poslužitelja, može se razvijati na strani klijenta izradom novih programčića (*eng.* Widget) u Javi, pa čak i čistih aplikacija na strani klijenta koje se pokreću isključivo u pregledniku. Vaadinov okvir za klijentski dio aplikacije uključuje Google Web Toolkit, koji pruža kompajler iz programskog jezika Java u JavaScript koji se pokreće u pregledniku. Tako Vaadin koristi programski jezik Java na oba dijela web aplikacije.

Karakteristike Vaadina:

- Spring podrška – podržava Spring 5 i Spring Boot 2, koristi Spring Boot postavke za konfiguraciju aplikacije
- Web komponente – razvijen prema W3C standardu web komponenti, ima Java API-je za sve komponente
- Java web razvoj – siguran razvoj web aplikacije samo u jeziku Java. Koristi Maven i Gradle alate za izgradnju aplikacije

5.3. Play

Play je programski okvir koji se koristi za izradu web aplikacija. To je okvir otvorenog kôda, što znači da je besplatan i da se može modificirati na različite načine. Play je pisan u programskim jezicima Java i Scala. Scala je programski jezik koji podržava i objektno-orijentirano i funkcionalno programiranje. Scala izvorišni kőd može se kompilirati u Java bajt kôd i pokretati na Java virtualnom stroju, a isto tako se može kompilirati u JavaScript i izvršavati u pregledniku. Gledajući objektno-orijentirani princip, Scala je jako slična Javi, ali Scala ima funkcionalni princip koji ga razlikuje od Jave. Značajke funkcionalnog programiranja koje Scala sadržava: nema razlike između izjava (*eng.* Statement) i izraza (*eng.* Expression), zaključavanje tipova (*eng.* Type inference), nepromjenjive (*eng.* Immutable) varijable i objekti.

Razvojna okruženja u kojima se može razvija web aplikacija pomoću programskog okvira Play su razvojna okruženja koja podržavaju Java ili Scala programski jezik, a to su: Eclipse, IntelliJ, NetBeans, ENSIME. NetBeans nema podršku za generiranje projekata, ali postoji Scala dodatak (*eng.* Plugin) koji pomaže kod razvijanja Scala aplikacije. Ostala razvojna okruženja imaju podršku za generiranje Scala projekata, ali je potrebno uključiti Scala dodatak.

Play programski okvir se koristi za izradu klijentskog i poslužiteljskog dijela aplikacije. Ima sve komponente koje su potrebne za izradu web aplikacije i REST servisa, kao što su: HTTP server, rad s obrascima, snažan mehanizam usmjeravanja (*eng.* Routing), I18n podršku za internacionalizaciju i lokalizaciju. Play je lagani okvir bez stanja, prilagođen webu koji koristi *Akku* kako bi pružao predvidljive i minimalne potrošnje resursa. Akka je besplatni skup alata otvorenog kôda koji smanjuje vrijeme izvođenja izgradnje i distribucije aplikacije na Java virtualnom stroju.

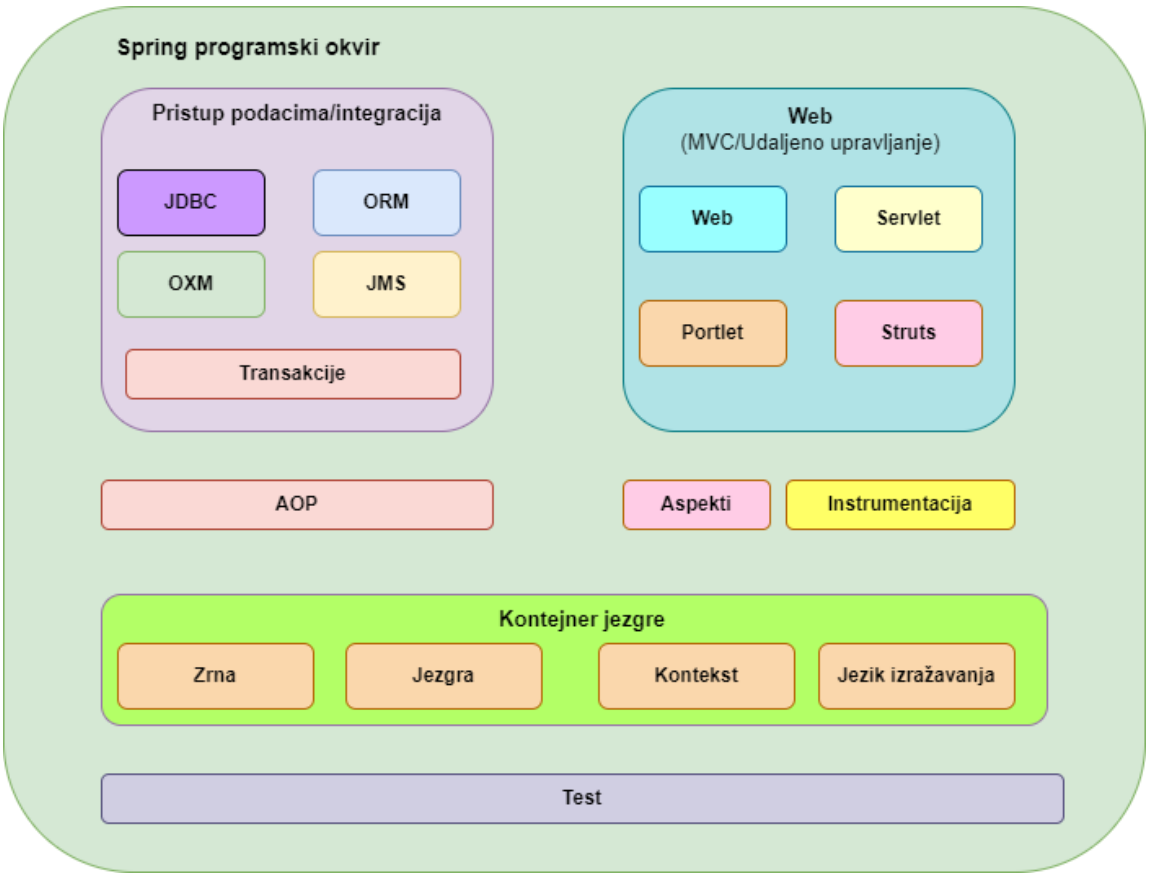
Karakteristike Playa:

- Koristi se za klijentski i poslužiteljski dio aplikacije, uključujući bazu podataka, sigurnost i internacionalizaciju
- Promjenom kôda, automatski se vidi promjena na aplikaciji
- Podrška za fleksibilne, skalabilne i aplikacije vođene događajima

6. Programski okvir Sping

Spring programski okvir služi za izradu srednjih i velikih web aplikacija koji koristi programski jezik Java. Spring je modularan, što znači da se sastoji od više modula. Aplikacija razvijena u Springu ne mora koristiti sve njegove module. Na primjer, za pristup podacima u bazi podataka može se koristiti JDBC modul ili Hibernate, ali ne moraju se oba. Prema Maneu i kolegama [15], ključni element Springa je infrastrukturna podrška na razini aplikacije: Spring se fokusira na spremanje poslovnih aplikacija tako da se razvojni programeri mogu usredotočiti samo na poslovnu logiku na razini aplikacije, bez nepotrebnih veza s određenim okruženjima za implementaciju. Spring uključuje:

- Fleksibilno uvođenje ovisnosti pomoću anotacijama
- Napredna podrška za aspektno orijentirano programiranje (*eng.* Aspect Oriented Programming; AOP)
- Podrška za uobičajene programske okvire otvorenog kôda kao što su Hibernate i Quartz
- Fleksibilni web okvir za izgradnju RESTful MVC aplikacije



Slika 9. Pregled modula programskog okvira Spring, prema dokumentaciji Springa [16]

Slika 9 prikazuje module programskog okvira Spring. Na slici se može primijetiti da se Spring programski okvir sastoji od 20ak modula koji su podijeljeni u sljedeće grupe: pristup podacima/integracija, web, AOP, aspekti, instrumentacije, kontejnera jezgre (*eng.* Core container) i test.

Kontejner jezgre sastoji se od zrna (*eng.* Beans), jezgre, konteksta i jezika izražavanja (*eng.* Expression Language), a ovdje slijede uloge tih modula:

- Jezgra – pruža temeljne dijelove programskog okvira, uključujući značajke inverzije kontrole (*eng.* Inversion of Control; IoC) i injekcije ovisnosti (*eng.* Dependecy Injection).
- Zrna – pruža *BeanFactory* koji je elegantna implementacija tvorničkog (*eng.* Factory) uzroka dizajna.
- Kontekst – temelji se na čvrstoj osnovi koju pružaju moduli jezgra i zrna i medij je za pristup svim definiranim i konfiguriranim objektima. Sučelje *ApplicationContext* je glavna točka kontekst modula.
- Jezik izražavanja – pruža moćan izrazni jezik za postavljanje upita i manipuliranjem grafom objekata tijekom izvođenja.

Pristup podacima/integracija sastoji se od JDBC, ORM, OXM, JMS i transakcije, a uloge tih modula su:

- JDBC – pruža JDBC apstrakcijski sloj koji uklanja potrebu za zamornim JDBC kôdiranjem
- ORM – pruža integracijske slojeve za popularne API-je za objektno relacijsko mapiranje, uključujući JPA, JDO, Hibernate i iBatis
- OXM – pruža sloj apstrakcije koji podržava implementacije mapiranja Objekt/XML za JAXB, Castor, XMLBeans, JiBX i XStream
- JMS – sadrži značajke za kreiranje i konzumaciju poruka
- Transakcije – podržava programsko i deklarativno upravljanje transakcijama za klase koje implementiraju posebna sučelja za sve POJO (*eng.* Plain old Java object)

Web modul sastoji se od web, Servlet, Portlet i Struts modula, a njihove uloge su:

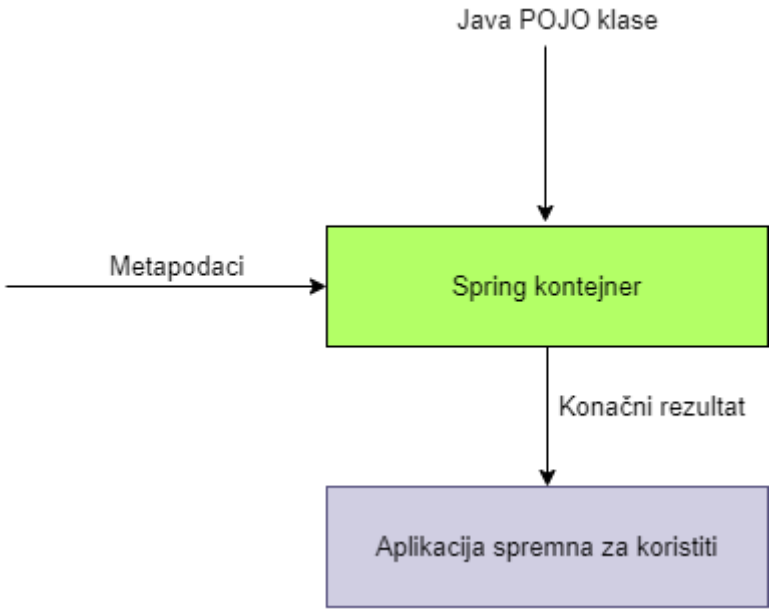
- Web – pruža osnovne web orijentirane integracijske značajke kao što su učitavanje višedijelne datoteke, inicijalizacija IoC kontejnera pomoću slušatelja (*eng.* Listener) *servleta* i web orijentiranog konteksta aplikacije
- Servlet – sadrži Spring MVC implementaciju za web aplikacije
- Struts – sadrži podupiruće klase za integraciju klasičnog Struts web sloja unutar Spring aplikacije
- Portlet – pruža MVC implementaciju koja se koristi u okruženju portleta i održava funkcionalnost web Servlet modula.

Ostali ne grupirajući moduli su AOP, aspekti, instrumentacija i test, a njihove uloge su:

- AOP – pruža implementaciju programiranja orijentiranu na aspekte koja omogućuje definiranje metoda presretača (*eng.* Inteceptor) koji trebaju jasno razdvojiti kôd koji implementira funkcionalnost koja bi trebala biti odvojena
- Aspekti – pružaju integraciju s AspectJ koji je snažan i zreo AOP okvir
- Instrumentacija – pruža podršku za instrumentaciju klase i implementaciju učitavača klase (*eng.* Class loader) koji se koristi u određenim aplikacijskim poslužiteljima
- Test – podržava testiranje Spring komponenti s okvirima JUnit ili TestNG

Prema Maneu i drugima [15], osnovni dio Spring okvira je Spring kontejner. Kontejner je zadužen za izradu, spajanje, konfiguriranje objekata te upravljanjem njihovim životnim ciklusom od kreiranja do uništenja. Spring kontejner koristi ubrizgavanje ovisnosti (*eng.* Dependecy Injection) kako bi upravljao komponentama koje izgrađuju aplikaciju. Ti objekti se zovu zrna (*eng.* Beans).

Kontejner dobiva upute o objektima koje treba instancirati, konfigurirati i sastaviti čitanjem konfiguracijskih metapodataka. Konfiguracijski metapodaci mogu biti u obliku Java kôda, Java anotacija ili u XML obliku. Na sljedećoj slici, dijagram prikazuje kako Spring radi. Spring IoC kontejner koristi Java POJO klase i konfiguracijske metapodatke za proizvodnju potpuno konfiguriranog izvršnog sustava ili aplikacije.



Slika 10. Spring IoC kontejner, prema Mane [15]

Postoje dva različita tipova takvih kontejnera: *Spring BeanFactory Container* i *Spring ApplicationContext Container*.

Spring BeanFactory Container je najjednostavniji kontejner koji pruža osnovnu podršku ubrizgavanju ovisnosti. *BeanFactory* i njegova povezana sučelja, kao što su *BeanFactoryAware*, *InitializingBean*, i *DisposableBean*, su dalje prisutni u Springu kako bi Spring bio kompatibilan s puno vanjskih programskih okvira koji su integrirani u Springu.

Spring AplicationContext Container ima više funkcionalnosti specifičnih za poduzeća kao što je mogućnost čitanje tekstualnih poruka iz datoteke svojstva (*eng.* Properties file) i mogućnost objavljivanja aplikacijskih događaja pomoću slušatelja događaja (*eng*. Event listeners).

AplicationContext kontejner ima sve funkcionalnosti kao i *BeanFactory* kontejner, pa je preporučljivo koristiti *AplicationContext* kontejner. *BeanFactory* se preporuča koristiti za lagane aplikacije koje zauzimaju manje prostora kao što su mobilne aplikacije i aplikacije bazirane na *apletima* gdje su količina podataka i brzina ključne.

Objekti koji čine okosnicu aplikacije i kojima upravlja Spring IoC kontejner se zovu zrna. Zrno je objekt koji je instanciran, sastavljen i njime se upravlja od strane Spring IoC kontejnera. Zrna su kreirana s konfiguracijskim metapodacima koji se dodaju kontejneru. Postoje tri načina kako Spring IoC kontejner može čitati konfiguracijske metapodatke:

- XML konfiguracijska datoteka
- Konfiguracija temeljena na anotacijama
- Konfiguracija temeljena na Javi

6.1. Spring Boot aplikacije

Spring Boot je programski okvir koji slijedi pristup „Konvencija iznad konfiguracije“ koji pomaže brzo i jednostavno razviti aplikaciju baziranu na Springu [17]. Glavni cilj Spring Boota je brzo kreiranje aplikacije bazirane na Springu bez toga da se od razvojnog programera zahtjeva da ponovno i ponovo piše konfiguraciju. Spring Boot je programski okvir baziran na javi i koristi se za kreiranje mikroservisa. Mikroservis definira pristup arhitekturi koja podijeli aplikaciju na više manjih servisa koji implementiraju poslovne zahtjeve.

Za razliku od Springa, Spring Boot se uglavnom koristi za razvijanje REST API-ja. Glavna razlika između njih je to što je glavna značajka Springa ubrizgavanje ovisnosti, dok je u Spring Bootu autokonfiguracija. Razvojnim programerima pomoću Spring Bootove autokonfiguracije je smanjeno potrebno vrijeme i trud za razvoj aplikacije te povećavaju produktivnost. Razlike između Springa i Spring Boota prikazane su u sljedećoj tablici.

Tablica 2. Razlike između Springa i Spring Boota, prema Srivastavi [18]

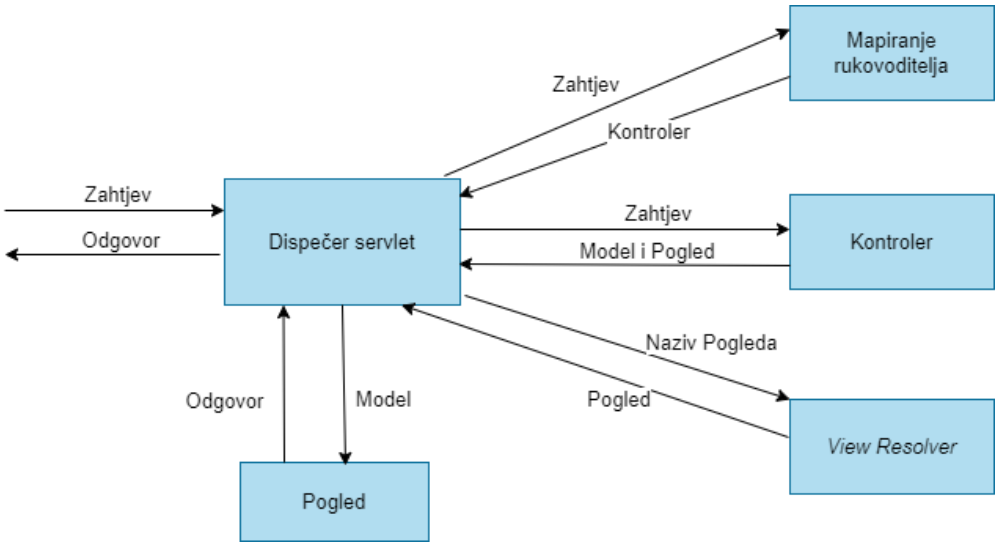
Spring	**Spring Boot**
Spring je programski okvir otvorenog kôda koji se široko koristi za razvoj poslovnih aplikacija	Spring Boot je izgrađen na temelju konvencionalnog Spring okvira koji se često koristi za izradu REST API-ja.
Najvažnija značajka Spring okvira je ubrizgavanje ovisnosti	Najvažnija značajka Spring Boota je autokonfiguracija
Pomaže u stvaranju labavo povezane aplikacije	Pomaže u izradi samostalne aplikacije
Da se pokrene Spring aplikacija, potrebno je eksplicitno postaviti poslužitelj	Spring Boot pruža ugrađene poslužitelje kao što su Tomcat i Jetty
Za pokretanje Spring aplikacije potreban je deskriptor postavljanja (*eng.* Deployment	Nema potrebe za deskriptorom postavljanja.

descriptor)	
Za izradu Spring aplikacije programeri pišu puno kôda	Smanjuje broj linija kôda
Ne pruža podršku za bazu podataka u memoriji	Pruža podršku za bazu podataka u memoriji kao što je H2

6.2. Spring MVC

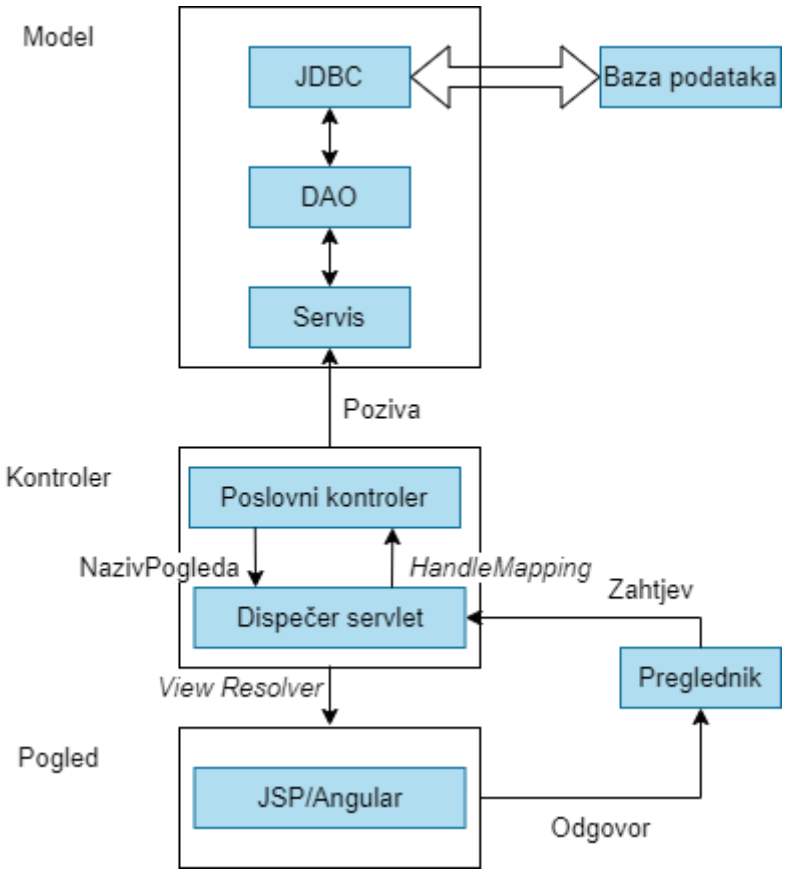
Spring MVC ima implementirani uzorak dizajna Model-Pogled-Kontroler (*eng. Model-View-Controller; MVC*) koji sustav dijeli na spomenuta tri sloja. Svaki sloj je relativno neovisan i svaki ima svoj zadatak. Spring MVC odvaja uloge kontrolera, modela, objekata, dispečera (*eng.* Dispatcher) i objekta rukovatelja (*eng. Handler objects*), zbog čega je lakše njima upravljati i mijenjati ih [19].

Na slici ispod prikazan je proces obrade zahtjeva kod Spring MVC-a. Dakle, kada klijent pošalje HTTP zahtjev, glavni kontroler, dispečer servlet (*eng.* Dispatcher servlet), prima zahtjev i traži Mapiranje rukovoditelja (*eng.* Handler Mapping), prema URL-u za slanje zahtjeva odgovarajućem kontroleru. Kontroler poziva odgovarajuću poslovnu logiku za obradu zahtjeva i kad se proces završi vraća ime pogleda i objekt *Model i Pogled* dispečer servletu. Na kraju se pomoću *ViewResolvera* vraćeni *Model i Pogled* prikazuje u odgovarajućem prikazu.



Slika 11. Spring MVC obrađivanje zathjeva, prema Zhangu [19]

Na slici 12. prikazana je struktura Spring MVC okvira. Prema slojevitoj ideji MVC okvira, MVC se sastoji od tri sloja: model, pogled, kontroler. Sloj kontrolera sadrži glavni kontroler (Dispečer servlet) i poslovni kontroler (*eng. Business Controller*). Sloj pogleda sadrži programski okvir pomoću kojeg se prikažu podaci na pregledniku kao što su JSP i Angular. Sloj modela sadrži objekt pristupa podacima (*eng.* Dana Access Object; DAO) sloj, sloj servisa i JDBC sloj koji koristi DAO sloj kako bi pristupio bazi podataka. Na slici ispod je osim slojeva prikazan i tijek obrada zahtjeva kroz slojeve. Klijent preko preglednika šalje zahtjev na sloj kontrolera, koji poziva sloj modela. Kada kontroler dobije potrebne podatke od modela, šalje podatke sloju pogleda koji prikazuje podatke na pregledniku.



Slika 12. Struktura Spring MVC okvira, prema Zhangu [19]

6.3. Spring web servisi

Spring web servisi (Spring-WS) je proizvod Spring zajednice usmjeren na stvaranje web usluga vođenih dokumentima [20]. Cilj Spring WS-a je olakšati razvoj SOAP usluge, omogućujući stvaranje fleksibilnih web usluga korištenjem jednog od mnogih načina za manipuliranjem XML sadržaja.

Glavne značajke Spring WS-a prema Poutsmi [20]:

- **Snažna mapiranja** - dolazne XML zahtjeve može se distribuirati bilo kojem objektu, ovisno o sadržaju poruke, zaglavlju SOAP akcije ili XPath izrazu.
- **XML API podrška** - dolaznim XML porukama može se rukovati ne samo standardnim JAXP API-jima kao što su DOM, SAX i StAX, već i JDOM, dom4j ili XOM.
- **Fleksibilni XML *marshalling*** - Spring WS temelji se na modulu Object/XML mapiranje u Spring okviru, koji podržava JAXB 1 i 2, Castor, XMLBeans, JiBX i XStream.
- **Ponovno upotreba Springa** - Spring-WS koristi kontekste Spring aplikacija za sve konfiguracije, što bi trebalo pomoći Spring programerima da brzo uđu u rad. Također, arhitektura Spring-WS-a nalikuje Spring-MVC-u.
- **Podržava WS-Security** - WS-Security omogućuje potpisivanje SOAP poruka, njihovo šifriranje i dešifriranje ili autentifikaciju prema njima.
- **Integrira se sa Spring Security** - WS-Security implementacija Spring WS pruža integraciju sa Spring Security - to znači da se postojeću konfiguraciju Spring Security može koristiti i za SOAP uslugu.
- **Apache licenca**.

Za implementaciju web servisa u Springu potrebno je na krajnjoj točki koristiti određene anotacije. Ispod slijedi tablica s potrebnim anotacijama za implementaciju Spring WS-a.

Tablica 3. Anotacije za implementaciju Spring WS-a, prema Poutsmi [20]

Anotacija	**Značenje**
@Endpoint	Registrira klasu sa Spring WS-om kao potencijalnog kandidata za obradu dolaznih SOAP poruka
@PayloadRoot	Koristi se za odabir metode rukovatelja, na temelju prostora imena (*eng.* Namespace) i lokalnog dijela
@RequestPayload	Označava da će dolazna poruka biti mapirana u parametar zahtjeva metode
@ResponsePaylod	Mapira vraćenu vrijednost u korisni teret (*eng. Payload*)

6.4. Spring RESTful

REST servisi su se već spominjali u prijašnjem poglavlju o web servisima. Također već se spomenulo da Spring Boot je namijenjen za brzu izradu REST API-ja te nema posebnog Spring modula za REST servise. U ovom poglavlju će biti navedene i objašnjene anotacije koje se koriste za izradu krajnje točke za REST servis. U tablici ispod prikazane su anotacije za izradu REST servisa.

Tablica 4. Anotacije za izradu REST servisa, prema Johnsonu i drugima [16]

Anotacija	**Značenje**
@RestController	Spoj @Controller i *@ResponseBody* anotacija, označava da će podaci koje vraća svaka metoda biti zapisani ravno u tijelo odgovara umjesto prikazivanje predloška
@RequestMapping	Mapira dobivene zahtjeve u klase i metode unutar kôda. Može se zamijeniti anotacijama *@GetMapping, @PostMapping, @PutMapping* i *@DeleteMapping* ovisno o metodi koja se treba implementirati
@ResponseBody	Omogućuje da Spring veže povratnu vrijednost metode za tijelo HTTP odgovora. Pretvara Spring objekt u JSON format.
@RequestParam	Spring raščlanjuje parametre zahtjeva i stavlja odgovarajuće u argumente metode.
@PathVariable	Mapira vrijednosti iz URL-a u odgovarajuće argumente metode.
@RequestBody	Za pristup tijelu HTTP zahtjeva. Sadržaj tijela pretvara se u tip argumenta deklarirane metode pomoću implementacija *HttpMessageConverter*

7. Docker – isporuka u kontejnerima

Docker je program koji se bavi kontejnerizacijom. Kontejnerizacija je oblik virtualizacije gdje se aplikacije izvode u izoliranim korisničkim prostorima, koji se nazivaju kontejneri, dok koriste isti zajednički operacijski sustav. U ovom poglavlju će biti više rečeno o samom Dockeru, objašnjena će biti kontejnerizacija kojom se bavi Docker te će na kraju biti usporedba Dockera i Podmana, aplikacije slične Dockeru.

7.1. Općenito o Docker-u

Tvrtku dotCloud osnovali su Kamel Founadi, Solomon Hykes i Sebastien Pahl 2008. godine u Parizu. Ta tvrtka je prethodnik tvrtke Docker Inc. koju su osnovali isti ljudi 2010. godine u Americi. Tvrtka Docker Inc. je razvila program Docker i prva stabilna verzija je bila objavljena 2013. godine. Docker, prema Turnbullu [21], je mehanizam otvorenog koda koji automatizira implementaciju aplikacija u kontejnere.

Kako bi se bolje razumjelo čime se Docker bavi i koja mu je svrha, potrebno je razumjeti što su to kontejneri u računalnom svijetu. Za razliku od hipervizor (*eng.* Hypervisor) virtualizacije, gdje se jedan ili više stroja pokreću virtualno na fizičkom sklopovlju (*eng.* Hardware) preko posredničkog sloja, kontejneri pokreću korisnički prostor na jezgri operacijskog sustava. Zbog toga se virtualizacija bazirana na kontejnerima često naziva i virtualizacija na razini operacijskog sustava. Tehnologija kontejnera omogućuje pokretanje više izoliranih korisničkih prostora na jednom računalu. Kontejneri imaju ograničenja, pa su tako za razliku od potpuno izolirane hipervizorske virtualizacije manje sigurni, odnosno skloni napadima. Ali unatoč ograničenjima, kontejneri se koriste u različitim slučajevima kao što su: implementacije velikih razmjera za usluge s više korisnika, lagani *Sandbox* (sigurnosni mehanizam za odvajanje pokrenutih programa) te kao okruženja za izolaciju procesa.

Koristeći nove tehnologije kontejnera kao što su: OpenVz, Solaris Zones i Linux kontejneri, kontejneri sada izgledaju kao potpuni domaćini (*eng.* Host), a ne kao okruženja za izvršavanje. Kod Dockera, posjedovanje modernih značajki jezgre Linuxa, kao što su kontrolne grupe i prostori imena, znači da kontejneri mogu imati snažnu izolaciju, vlastitu mrežu i hrpe za pohranu, kao i mogućnosti upravljanja resursima kako bi se omogućila koegzistencija više kontejnera na domaćinu.

Kontejneri se smatraju ekonomičnom tehnologijom jer zahtijevaju ograničene troškove. Za razliku od tradicionalnih tehnologija virtualizacije, oni ne zahtijevaju sloj emulacije ili sloj hipervizora za rad i umjesto toga koriste normalno sučelje za pozive operacijskog sustava. To smanjuje opterećenje potrebno za pokretanje spremnika i može omogućiti veću gustoću kontejnera za izvođenje na glavnom računalu.

7.2. Docker – tehnologija kontejnera

Docker dodaje mehanizam za implementaciju aplikacije iznad izvršnog okruženja virtualnog kontejnera. Dizajniran je za pružanje laganog i brzog okruženja za pokretanje kôda, kao i za učinkovit tijek rada za prijenos kôda s računala na testno okruženje i proizvodnju. Docker je jako jednostavan, za njegovo pokretanje dovoljno je da računalo ima kompatibilnu Linux jezgru i Docker binarnu datoteku. Prema Turnbullu [21], Dockerova misija je pružiti:

Jednostavan i lagan način modeliranja stvarnosti

Docker je brz pa tako aplikacija se može kontejnerizirati za nekoliko minuta. Oslanja se na model kopiraj na napisano (*eng.* Copy-on-write) pa je svako unošenje promjena u aplikaciju isto vrlo brzo.

Logična podjela dužnosti

Razvojnim programerima dovoljno je da se brinu o samoj aplikaciji koja se izvodi unutar kontejnera, dok o upravljanju kontejnerima brine Docker. Također, Docker je osmišljen kako bi poboljšao dosljednost osiguravajući da okruženje u kojem programer piše kôd odgovara okruženjima u kojima su aplikacije raspoređene.

Brz, učinkovit životni ciklus razvoja

Dockerov cilj je smanjiti vrijeme ciklusa između kôda koji se piše, kôda koji se testira, implementira i koristi, odnosno cilj mu je aplikacije učiniti prenosivima i lakima za izradu.

Potiče arhitekturu usmjerenu na usluge

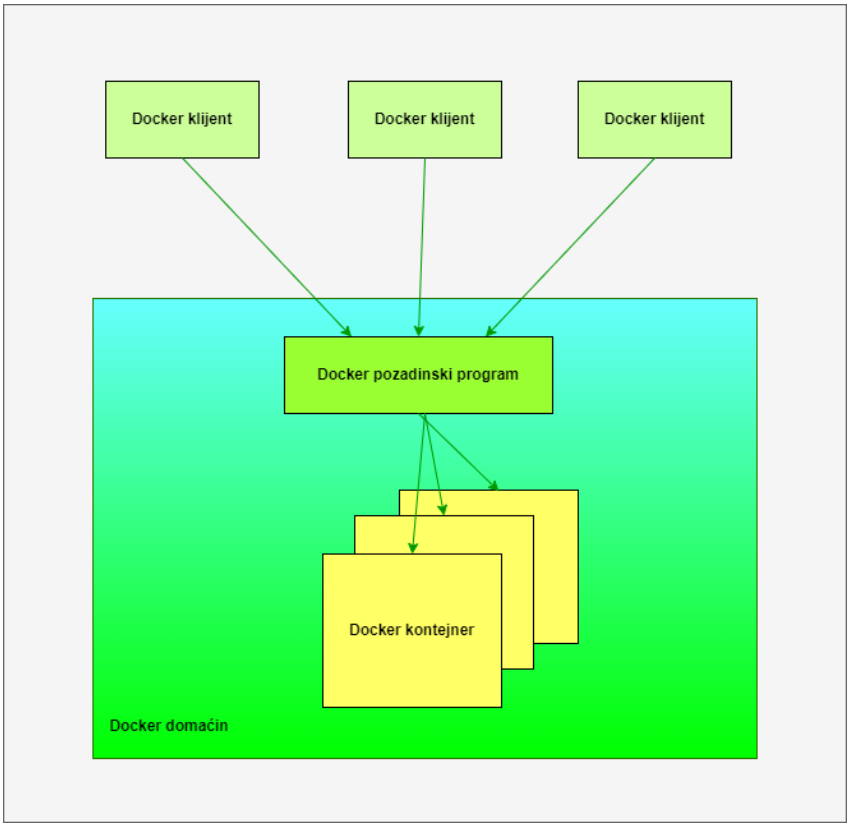
Docker potiče servisno orijentirane i mikroservisne arhitekture. Preporučeno je da svaki kontejner pokreće jednu aplikaciju ili proces. Tako se promiče distribucija aplikacije gdje je aplikacija ili servis predstavljen nizom međusobno povezanih kontejnera. To olakšava distribuciju, skaliranje i uklanjanje pogrešaka kod aplikacija.

Prema Turnbullu [21], Docker se sastoji od četiri glavnih komponenta koje sačinjavaju sam Docker:

- Docker klijent i poslužitelj
- Docker slike (*eng.* Images)
- Registri
- Docker kontejneri

Docker klijent i poslužitelj

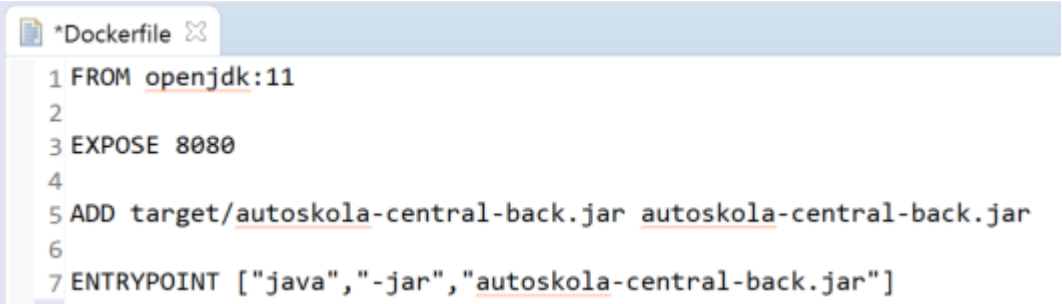
Docker je aplikacija koja koristi klijent-poslužitelj arhitekturu. Docker klijent komunicira s Docker poslužiteljem ili pozadinskim programom (*eng.* Daemon), koji zauzvrat obavlja sav posao. Docker se isporučuje binarno pomoću naredbenog retka ili Dockerom. Docker pozadinski program i klijent mogu se pokrenuti na istom domaćinu ili povezati lokalni Docker klijent s udaljenim Docker pozadinskim programom koji radi na drugom domaćinu. Dockerova arhitektura prikazana je na slici ispod.



Slika 13. Arhitektura Dockera, prema Turnbullu [21]

Docker slike

Docker slike su glavni „građevni“ blokovi kod Dockera. Kontejneri se pokreću iz Docker slika. Slike su dio životnog ciklusa Dockera koji se zove izgradnja (*eng.* Build). One su slojeviti format, koji koristi Union datotečni sustav, koji se izgrađuje korak po korak koristeći niz uputa.



Slika14. Primjer Docker slike, [Autorski rad]

Docker slike mogu se smatrati izvornim kôdom za kontejnere. One su prenosive i mogu se dijeliti, pohranjivati i ažurirati. Na slici iznad, prikazan je primjer niza uputa Docker slike. Mogu se primijetiti četiri naredbe: *from, expose, add* i *entrypoint*. *From* specificira osnovnu sliku, *expose* izlaže vrata (*eng.* Port), *add* kopira datoteke i direktorije u kontejner i *entrypoint* daje naredbu i argumente za izvršni kontejner.

Registri

Docker pohranjuje slike koje se izgrade u registre. Postoje dvije vrste registara: javni i privatni. Docker Inc. upravlja javnim registrom za slike koji se naziva Docker Hub. Docker Hub sadrži preko 10 000 slika koje su napravili drugi ljudi i podijelili. Također sadrži slike poput *Nginx* web poslužitelja ili MySQL baze podataka. U privatni registar se mogu pohraniti vlastite slike koje mogu uključivat izvorni kôd ili druge vlasničke informacije koje se žele zaštiti ili podijeliti samo sa određenim ljudima.

Docker kontejner

Docker pomaže izgraditi i implementirati kontejnere unutar kojih se mogu pakirati aplikacije i servisi. Kao što je prije navedeno, kontejneri se pokreću iz slika i mogu sadržavati jedan ili više pokrenutih procesa. Docker kontejner je:

- Format slike
- Skup standardnih operacija
- Izvršno okruženje

Docker posuđuje koncept standardnog brodskog kontejnera, koji se koristi za globalni transport robe, kao model za svoje kontejnere. Ali umjesto dostave robe, Docker kontejneri šalju softver. Svaki Docker kontejner sadrži sliku programa, kod brodskog kontejnera teret, i nad njim se omogućuje izvođenje različitih operacija. Može se stvoriti, pokrenuti, zaustaviti, ponovo pokrenuti i uništiti. Kao i kod brodskog kontejnera, Dockeru nije bitan sadržaj kontejnera kada izvršava navedene operacije, odnosno Dockeru nije bitno je li kontejner web poslužitelj ili baza podataka, on svaki kontejner tretira jednako. Također Dockeru nije bitno ni kamo se šalje kontejner: može se graditi na vlastitom računalu, učitati u registar, preuzeti na fizički i virtualni poslužitelj, itd. Baš poput normalnog kontejnera, Dockerov kontejner je izmjenjiv, složiv i prenosiv.

Turnbull [21] navodi neke primjere za što i gdje se Docker može primijeniti:

- Docker pomaže da lokalni razvoj i izgradnja tijeka rada aplikacije bude brži, učinkovitiji i lakši. Lokalni programeri mogu graditi, pokretati i dijeliti Docker kontejnere. Kontejneri mogu biti izgrađeni u razvoju i potom se promovirati u okruženja za testiranje te na kraju i u proizvodnju.
- Pokretanje samostalnih servisa i aplikacija kroz više okruženja, koncept koji je posebno koristan u arhitekturama orijentiranim na servise i implementacije koje se oslanjaju na mikroservise.
- Izrada i testiranje složenih aplikacija i arhitektura na lokalnom računalu prije postavljanja u produkcijsko okruženje
- Izgradnja višekorisničke infrastrukture PaaS (*eng.* Platform-as-a-Service)
- Visokoučinkovite, implementacije domaćina velikih razmjera.

7.3. Usporedba Dockera i Podmana

U ovom poglavlju će se usporediti dva slična programa za kontejnerizaciju: Docker i Podman. Budući da je već dosta rečeno o Dockeru, ovdje će ukratko biti opisan Podman kako bi ih se moglo usporediti. Podman se, kao i Docker, bavi kontejnerima, otvorenog je kôda te za razliku od Dockera nema pozadinskog programa (*eng*. Daemonless). Koristi Linuxov izvorni alat koji olakšava pronalaženje, izgradnju, dijeljenje i implementaciju aplikacija pomoću *Open Containers Initiative* (OCI) kontejnera i slika kontejnera. Kontejneri koji su pod kontrolom Podmana mogu pokrenuti korijenski (*eng.* Root) ili neprivilegirani korisnik. Podman upravlja cijelim sustavom kontejnera koji uključuje Podove (*eng.* Pods), kontejnera i slike kontejnera. Podman ima modularan dizajn koji mu omogućuje da koristi

individualne komponente sustava samo kad ih treba. Uspredba Dockera i Podmana je pokazana u tablici ispod.

Tablica 5. Usporedba Dockera i Podmana, prema Aleksiću [22]

	Docker	**Podman**
Pozadinski program	Koristi pozadinski program	Ima arhitekturu bez pozadinskog programa
Korijen	Kontejnere mogu pokretati samo korijenski korisnici	Kontejnere mogu pokretati korijenski i nekorijenski korisnici
Slike	Može izgraditi slike za kontejner	Koristi Buildah za izgradnju slike
Monolitna platforma	Da	Ne
Docker-swarm	Podržava	Ne podržava
Docker-compose	Podržava	Podržava
Pokreće se na	Linux, macOS, Windows	Linux, macOS, Windows (sa WSL-om)

Kao što se može prijetiti u tablici iznad, razlika između Dockera i Podmana je ta da Podman ne koristi pozadinski program te da kod Podmana osim korijenskih korisnika, kontejnere mogu pokretati i nekorijenski korisnici. S druge strane Docker može sam izgraditi slike za kontejner, dok Podman mora koristiti Buildah za izgradnju slike. Docker je monolitna platforma i podržava Docker-swarm, alat za grupiranje i raspoređivanje kontejnera. Obje platforme podržavaju Docker-compose, alat koji je razvijen za pomoć pri definiranju i dijeljenju aplikacija s više kontejnera, te se obje platforme mogu pokretati na Linuxu, macOSu te Windowsu.

8. Orkestracija kontejnera pomoću Kubernetes

Kubernetes je program koji se bavi orkestracijom kontejnera. Najčešće se Kubernetes povezuje s Dockerom pa se radi o Dockerovim kontejnerima. U ovom poglavlju će biti nešto više rečeno o Kubernetesu te o samoj orkestraciji kontejnera, odnosno koji su elementi Kubernetesa i kako se pomoću njih vrši orkestracija. Na kraju poglavlja će biti usporedba Kubernetesa s nekom sličnom tehnologijom.

8.1. Općenito o Kubernetes

Kubernetes je izvorno razvio Google, a kasnije ga preuzeo i održava *Cloud Native Computing Foundation* (CNFC). Napisan je u programskom jeziku Go te služi kao rešenje za učinkovitu implementaciju, upravljanje i skaliranje kontejnerske aplikacije u podatkovnim centrima u oblaku (*eng.* Cloud). Budući da je Kubernetes projekt otvorenog kôda, može biti konfiguriran i modificiran tako da bude temelj za neke druge platforme.

Kao što smo već naveli Kubernetes se bavi orkestracijom kontejnera, ali Poulton [23] to definira malo šire te kaže da Kubernetes zapravo orkestrira aplikacije. Većinom orkestrira kontejnerizirane mikroservisne aplikacije u oblaku. Ova rečenica zvuči malo zbunjujuće pa će se sada bolje objasniti značenje svake riječi.

Orkestrator je sustav koji postavlja aplikacije i upravlja njima. Može implementirati aplikacije i dinamički reagirati na promjene. Na primjer, Kubernetes može:

- Postaviti (*eng.* Deploy) aplikaciju
- Dinamički povećavati i smanjivati aplikaciju po potrebi
- Samoizliječiti (*eng* Self-healing) kada se nešto pokvari
- Izvoditi ažuriranja i vraćati na staro stanje bez prekida rada

Najbolja stvar je da Kubernetes može sve to obavljati bez nadzora i bez da se netko uključuje u njegov rad. Potrebno je samo na početku podesiti željene konfiguracije, a zatim Kubernetes radi sve sam.

Kontejnerizirana aplikacija je aplikacija koja se pokreće u kontejneru. Prije kontejnera aplikacije su se pokretale na fizičkim poslužiteljima ili na virtualnim strojevima. Kontejneri su brži, lakši i bolje opremljeni za poslovne zahtjeve. Kubernetes može orkestrirati različite vrste ranog opterećenja (*eng.* Workload), uključujući virtualne strojeve i funkcije bez poslužitelja, ali najčešće se koristi za orkestriranje aplikacija u kontejnerima.

Izvorna aplikacija u oblaku je aplikacija koja je dizajnirana da zadovolji moderne poslovne zahtjeve poput automatskog skaliranja, samopopravljanja, ažuriranja, te može raditi na Kubernetesu. Važno je za napomenuti da se izvorne aplikacije u oblaku ne moraju izvoditi samo u javnom oblaku, već se mogu i lokalno, odnosno bilo gdje postoji Kubernetes.

Mikroservisna aplikacija je poslovna aplikacija koja je izgrađena od više malih specijaliziranih dijelova koji komuniciraju i tvore smislenu aplikaciju. Na primjer, aplikacija se sastoji od sljedećih malih specijaliziranih komponenata:

- Web korisnički dio
- Servis za autentifikaciju
- Servis za prijavljivanje
- Baza podataka
- Itd.

Svaki od tih individualnih servisa se naziva mikroservis. Svaki taj mikroservis je neovisan o drugima, što znači da svaki mikroservis može programirati drugi tim te da se svaki može skalirati neovisno ostalima.

8.2. Razumijevanje orkestracije kontejnera

Kubernetes na najvišoj razni može biti ili klaster za pokretanje aplikacija ili orkestrator za izvorne mikroservisne aplikacije u oblaku. Kubernetes kao klaster je skup čvorova (*eng.* Nodes) i kontrolna ravnina (*eng.* Control Plane). Kontrolna ravnina izlaže API, ima planer (*eng.* Scheduler) za dodjelu poslova čvorovima, a stanje se bilježi u trajnoj pohrani. Čvorovi su mjesto gdje se pokreću aplikacijski servisi. Kontrolna ravnina se može smatrati mozgom zbog toga što implementira sve glavne značajke poput automatskog skaliranja i ažuriranje bez prekida rada. Dok se čvorovi mogu smatrati mišićima jer oni cijeli dan izvršavaju kôd aplikacije. Kubernetes kao orkestrator je sustav koji se brine postavljanju i upravljanju aplikacijama.

Kako bi se pokrenula aplikacija pomoću Kubernetesa, potrebno je prvo izrađenu aplikaciju kontejnerizirati i predati taj kontejner Kubernetes klasteru. Klaster se sastoji od jednog ili više glavnih čvorova (*eng.* Head nodes/masters) i puno običnih čvorova. Glavni čvor upravlja cijelim klasterom, što znači da je odgovoran za planiranje odluka, obavljanje nadzora, implementacije promjena i odgovaranje na događaje. Zbog toga se glavni čvor često poistovjećuje s kontrolnom ravninom. Obični čvorovi su mjesto gdje se pokreću aplikacije pa se zbog toga još nazivaju i podatkovnom ravninom (*eng.* Dana plane). Svaki čvor je povezan s glavnim čvorom te ga izvješćuje o svojem stanju i stalno čeka nove zadatke. Kako bi se pokrenula aplikacija na Kubernetesu potrebno je sljedeće:

1. Napisati aplikaciju kao male neovisne mikroservise u bilo kojem programskom jeziku
2. Zapakirati svaki mikroservis u kontejner
3. Zapakirati svaki kontejner u svoj Pod
4. Implementirati Podove u klaster putem kontrolera više razine kao što su *Deployment, DaeomnSets, StatefulSets*, itd.

Kao što je prije navedeno, klaster se sastoji od glavnih čvorova i čvorova. Glavni čvor je kolekcija sistemskih servisa koji čine kontrolnu ravninu klastera. Sistemski servisi koji čine glavnu ravninu su: API poslužitelj, pohrana klastera (*eng.* Cluster store), planer i kontroler.

API poslužitelj je glavna, centralna stanica Kubernetesa. Sva komunikacija između svih komponenti, vanjskih i unutranjih, mora proći kroz API poslužitelj. On izlaže RESTful API preko HTTPSa koji se definira u YAML konfiguracijski datoteci. YAML datoteke sadrže željeno stanje aplikacije, a pod željeno stanje aplikacije smatra se slika kontejnera koja će se koristiti, koje mrežne priključke izložiti i koliko Pod replika pokrenuti. Svi zahtjevi prema API poslužitelju podliježu provjerama provjere autentičnosti i autorizacije, ali nakon što se one obave, konfiguracija u YAML datoteci se provjerava, zadržava u spremištu klastera i postavlja na klaster.

Jedini je dio kontrolne ravnine koji prati stanje i trajno pohranjuje cijelu konfiguraciju i stanje klastera. Kao takav je vitalan dio klastera, jer bez pohrane nema klastera. Pohrana klastera trenutno se temelji na *etcd*, popularnoj distribuiranoj bazi podataka. Što se tiče dostupnosti, etcd preferira dosljednost u odnosu na dostupnost. To znači da će zaustaviti ažuriranje klastera po potrebi samo kako bi se održala dosljednost. Međutim, ako etcd postane nedostupan, aplikacije koje se izvode na klasteru trebale bi nastaviti raditi, samo

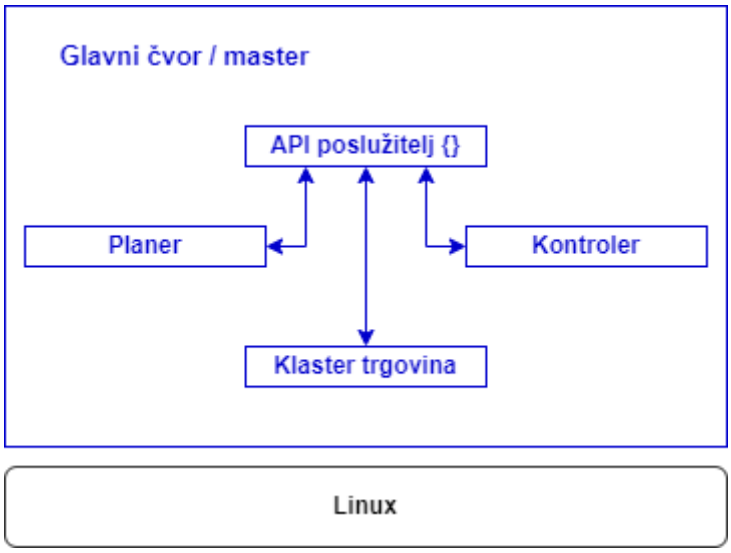
nećete moći ništa ažurirati. Kao i kod svih distribuiranih baza podataka, dosljednost pisanja u bazu podataka je vitalna.

Upravitelj kontrolera (*eng.* Controller manager) implementira sve pozadinske kontrolne petlje koje nadziru klaster i odgovaraju na događaje. To je upravljač nad kontrolerima, što znači da pokreće sve neovisne upravljačke petlje i nadzire ih. Kontrolne petlje uključuju: kontroler čvora, kontroler krajnjih točaka i kontroler skupa replika. Svaki od njih radi kao pozadinska petlja za promatranje koja neprestano prati API poslužitelj radi promjena. Cilj je osigurati da trenutno stanje klastera odgovara željenom stanju. Logika koju implementira svaka kontrolna petlja je sljedeća:

1. Postići željeno stanje
2. Promatrajte trenutno stanje
3. Odrediti razlike

Svaka kontrolna petlja je specijalizirana i zainteresirana samo za svoj zadatak Kubernetes klastera i nije ih briga za ostale kontrolne petlje. Ovo je ključno za distribuirani dizajn Kubernetesa i pridržava se Unixove filozofije izgradnje složenih sustava od malih specijaliziranih dijelova.

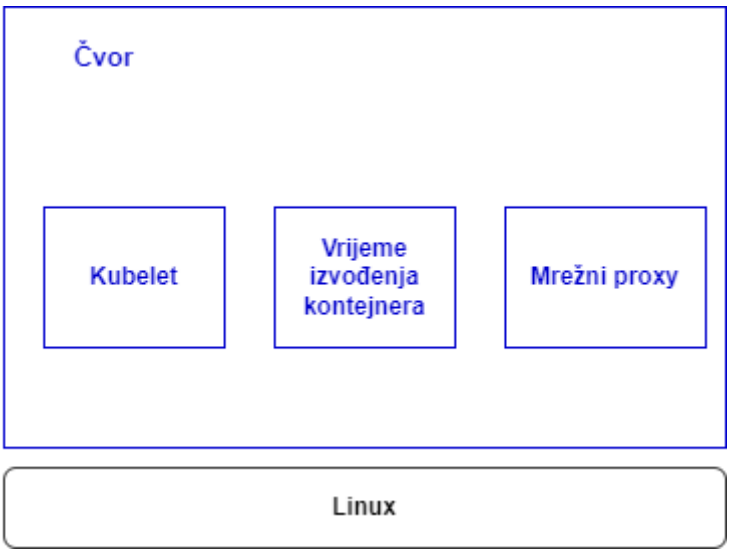
Planer promatra API poslužitelj tražeći nove radne zadatke i dodjeljuje ih odgovarajućim zdravim čvorovima. U pozadini implementira složenu logiku koja filtrira čvorove koji nisu u stanju izvršiti zadatak, a zatim rangira čvorove koji su sposobni. Prilikom identificiranja čvorova koji mogu pokrenuti zadatak, planer izvodi različite provjere predikata. Svaki čvor koji nije u stanju pokrenuti zadatak se zanemaruje, a preostali čvorovi se rangiraju prema stvarima kao što su: ima li čvor već potrebnu sliku, koliko slobodnih resursa ima čvor, koliko zadataka čvor već izvodi. Ako planer ne može pronaći odgovarajući čvor, zadatak se ne može rasporediti i označava se kao na čekanju. Planer nije odgovoran za pokretanje zadataka, već samo za odabir čvorova na kojima će se zadatak izvoditi.



Slika 15. Prikaz glavnog čvora klastera, prema Poultonu [23]

Čvorovi su zapravo radnici u Kubernetes klasteru te oni čine tri stvari:

1. Prate API poslužitelj za nove radne zadatke
2. Izvršavaju nove radne zadatke
3. Izviješćuju kontrolnu razinu (putem API poslužitelja)



Slika 16. Prikaz čvora klastera, prema Poultonu [23]

Na slici iznad se može vidjeti da je čvor klastera jednostavnije građe od glavnog čvora. Čvor se sastoji od tri elementa, a to su kubelet, vrijeme izvođenje kontejnera (*eng.* Container runtime; CRI) i mrežnog proxy-ija.

Kubelet je glavni Kubernetes agent i radi na svakom čvoru u klasteru. Kada se pridruži novi čvor klasteru, proces instalira kubelet na čvor. Kubelet je odgovoran za registraciju čvora u klasteru. Registracija učinkovito spaja procesor, memoriju i pohranu čvora u širi skup klastera. Jedan od glavnih zadataka kubeleta je promatranje API poslužitelja za nove zadatke. Kad god vidi novi zadatak, izvršava ga i održava kanal za izvještavanje natrag do kontrolne razine. Ako kubelet ne može pokrenuti određeni zadatak, javlja se glavnom uređaju i dopušta kontrolnoj razini da odluči koje radnje poduzeti.

Vrijeme izvođenje kontejnera je komponenta čvora koja je zadužena za zadatke povezane s kontejnerima kao što su: povlačenje slika, pokretanje i zaustavljanje kontejnera. CRI maskira unutarnju mašineriju Kubernetesa i izlaže čisto dokumentirano sučelje na koje se mogu priključiti vrijeme izvođenja (*eng.* Runtime) kontejnera treće strane.

Mrežni proxy radi na svakom čvoru u klasteru i odgovoran je za lokalno umrežavanje klastera. On osigurava da svaki čvor dobije vlastitu jedinstvenu IP adresu i implementira lokalna IPTABLES ili IPVS pravila za obrađivanje usmjeravanja (*eng.* Routing) i balansiranje opterećenja prometa na Pod mreži.

Kao i svaka komponenta kontrolne ravnine i čvora, tako i svaki Kubernetes klaster ima svoj interni DNS servis. DNS servis klastera ima statičnu IP adresu koja je zapisana u svaki Pod na klasteru pa zbog toga svi kontejneri i Podovi znaju kako pronaći tu IP adresu. Svaki novi servis automatski se registrira na klasterov DNS tako da sve komponente u klasteru mogu naći svaki servis po imenu. Neke druge komponente koje su registrirane s DNS-om klastera su *StatefulSetovi* i pojedinačni Podovi kojima on upravlja.

8.3. Usporedba Kubernetesa sa sličnim tehnologijama

U ovom poglavlju će se usporediti slične tehnologije za orkestraciju kontejnera. Tehnologije koje će se usporediti su: Kubernetes, Docker Swarm, Apache Mesos i Nomad. Budući da je već dosta rečeno o Kubernetesu, u ovom poglavlju će se kratko predstaviti ostale tehnologije te će se one tablično usporediti po određenim značajkama bitnih za tehnologije za orkestraciju kontejnera.

Docker Swarm je platforma za orkestraciju kontejnera otvorenog koda koju je izgradio i održava Docker. Docker Swarm pretvara više Docker instanci u jedno virtualno računalo. Docker Swarm klaster općenito sadrži tri stavke:

- Čvorovi
- Službe i zadaci
- Balanseri opterećenja

Čvorovi su pojedinačne instance Docker mehanizma koji kontroliraju klaster i upravljaju kontejnerima koji se koriste za pokretanje servisa i zadataka. Docker Swarm klasteri također uključuju balansiranje opterećenja za usmjeravanje zahtjeva preko čvorova.

Apache Mesos je projekt otvorenog kôda za upravljanje računalnim klasterima. Razvijen je na kalifornijskom sveučilištu Berkeley. Izgrađen je korištenjem istih principa kao Linux jezgra, samo na različitoj razini apstrakcije. Mesos kernel radi na svakom računalu i pruža aplikacije s API-jima za upravljanje resursima i raspoređivanje u cijelom podatkovnom centru i okruženjima oblaka.

Nomad program koji služi kao upravitelj klastera i planer dizajniran za mikroservise i skupna radna opterećenja kojeg je razvila tvrtka HashiCorp iz San Francisca. Nomad je distribuiran, visoko dostupan i skaliran na tisuće čvorova koji obuhvaćaju više podatkovnih centara i regija.

Ozmen [24] je analizirao sva četiri alata za orkestraciju kontejnera te napravio usporedbu po važnim značajkama za alate koje se bave orkestracijom kontejnera te je ta usporedba prikazana u tablici ispod.

Tablica 6. Usporedba alata za orkestraciju kontejnera, prema Ozmenu [24]

Značajka	**Docker Swarm**	**Kubernetes**	**Apache Mesos**	**Nomad**
Podržava kontejnere	Samo Docker kontejnere	Docker, Rkt	Docker, Rkt	Docker, Rkt, LXC
Otkrivanje servisa	Da	Da	Da	Program treće strane (*eng. Third party)* (Consul)
Repliciranje	Da	Da	Program treće	Da

glavnog čvora			strane (Zookeeper)	
Balansiranje opterećenja	Podržava vanjsko balansiranje	Podržava vanjsko balansiranje	Program treće strane (Marathon)	Program treće strane (Consul)
Automatsko skaliranje	Ne	Da	Da	Ne
Praćenje kontejnera	Program treće strane	Program treće strane	Program treće strane	Da
Bilježenje (*eng*. Logging) kontejnera	Program treće strane (ELK)	Program treće strane (ELK)	Da	Da
Upravljanje tajnama (*eng.* Secrets)	Da	Da	Ne	Program treće strane (Vault)

Kao što se može primijetiti iz tablice iznad, svi alati su dosta slični. Pa tako Docker Swarm može podržavati samo Dockerove kontejnere, dok ostali mogu uz Docker konetjenre podržavati i *Rkt* kontejnere, a Nomad može uz to i *LXC* kontejner. Svi alati mogu otkrivati svoje servise, ali Nomad za to koristi program treće strane. Što se tiče repliciranja glavnog čvora, svi ga mogu replicirati, ali Apache Mesos za to koristi program Zookeepr. Docker Swarm i Kubernetes podržavaju vanjsko balansiranje, dok Apache Mesos za to koristi Marathon, a Nomad koristi program Consul. Automatsko skaliranje imaju samo Kubernetes i Mesos, dok praćenje kontejnera ima samo Nomad, a ostali koriste program treće strane. Svi mogu bilježiti zapise kontejnera, ali Docker Swarmu i Kubernetesu za to treba program ELK. Apache Mesos jedini ne može upravljati tajnama, dok Nomad za to koristi Vault.

9. Aplikacija za vođenje autoškole

Ovo poglavlje prati razvoj praktičnog dijela rada, odnosno razvoj aplikacije za vođenje autoškole pomoću programskih okvira Angular za klijentski dio i Springa za poslužiteljski dio aplikacije. Nakon samog razvoja aplikacije slijedi postupak kontejnerizacije aplikacije te pokretanje aplikacije pomoću Kubernetesa.

9.1. Opis aplikacije [2]

Ideja je aplikacije, kao što sam naziv govori, vođenje autoškole preko aplikacije, drugim riječima informatizacija proces pohađanja i vođenja autoškole. Glavni koraci informatizacije ovih procesa su rješavanje ispita iz propisa i vođenje dnevnika vožnje. Polaznik autoškole može rješavati ispite iz propisa te nakon položenog ispita može odabrati instruktora. Nakon što instruktor dobije polaznike, on može voditi njihov dnevnik vožnje te će imati opciju da nakon minimalnog broja sati vožnje može, ukoliko student uspješno položi ispit iz vožnje, promijeniti status polaznika u status koji označava da je polaznik uspješno položio ispit.

Aplikacija ima četiri uloge: neregistrirani korisnik, registrirani korisnik, moderator i administrator. Svi korisnici mogu sve što i neregistrirani korisnik. Svaka uloga ima drugačiji pogled na aplikaciju pa samim time omogućene su im drugačije akcije.

Opis radnji je sljedeći:

•**Neregistrirani korisnik:** pregled osnovnih informacija o autoškoli, pregled vozila koje koristi autoškola, pregled instruktora vožnje, pogled na često postavljana pitanja

•**Registrirani korisnik:** Mogućnost prijave na određeni termin za određenu kategoriju vozila, praćenje svog statusa, prijavljivanje određenom instruktoru vožnje koji ima slobodnih mjesta, praćenje dnevnika odvoženih sati, rješavanje ispita iz propisa

•**Moderator (instruktor autoškole):** Praćenje statusa registriranih korisnika kojima je on instruktor, vođenje dnevnika odvoženih sati

•**Administrator** : pratiti dnevnik rada, pratiti rad instruktora, uređivanje konfiguracija, uređivanje osnovnih informacija o autoškoli

Arhitektura sustava sastoji se od tri dijela: baza podataka, poslužiteljski i klijentski dio. Kao što je prikazano na slici ispod, klijentski dio može komunicirati samo sa poslužiteljski dijelom, poslužiteljski dio komunicira sa bazam podataka i klijentskim dijelom, a baza podataka može komunicirati samo sa poslužiteljski dijelom. Klijentski dio aplikacije razvijen je u programskom okviru Angular, te njegova vrata imaju vrijednost *31000*. Poslužiteljski dio je razvijen u Springu, te njegova vrata imaju vrijednost *30163*, a baza podataka je MySQL baza podataka te njezina vrata imaju vrijednost *3306*. Vrijednosti vrata su definira *yml/yaml* datotekama koje koristi Kubernetes, a o njima će se pričat nešto kasnije u tekstu.

Slika 17. Arhitektura aplikacije za vođenje autoškole [Autorski rad]

9.2. Baza podataka i ERA dijagram [3]

Baza podataka je izrađena u programskom alatu MySQL Workbrench te se ona sastoji od 16 tablica. ERA dijagram prikazan je na slici 18, a popis tablica, njihovih ključnih atributa i opis slijede dalje u tekstu:

FAQ
FAQ_ID – primarni ključ
PITANJE – tekstualno polje koje sadrži tekst pitanja
ODGOVOR – tekstualno polje koje sadrži tekst odgovora

FAQ tablica služi za spremanje pitanja i odgovora na često postavljenja pitanja.

AUTOSKOLA_INFO
AUTOSKOLA_INFO_ID – primarni ključ
SIFRA – tekstualno polje koje sadrži šifru atributa (npr. adresu, kontakt broj, itd.)
VRIJEDNOST – tekstualno polje koje sadrži vrijednost određenog atributa

AUTOSKOLA_INFO tablica služi za spremanje osnovnih informacija o autoškoli

PITANJE

PITANJE_ID – primarni ključ

TEKST_PITANJA – tekstualno polje gdje se sprema tekst pitanja

SLIKA – tekstualno polje gdje se sprema *Base64* vrijednost slike

RASKRIZJE – vrijednost 1 ako je pitanje o raskrižju, 0 ako nije

BROJ_BODOVA – vrijednost koja označava broj bodova koje donosi pitanje

Tablica PITANJE služi za spremanje pitanja u bazu podataka.

ODABIR

ODABIR_ID – primarni ključ

PITANJE_ID – vanjski ključ na tablicu PITANJE, označava pitanje na koje je vezan odabir

TEKST – tekstualno polje koje sadrži tekst odabira

TOCAN_ODGOVOR – vrijednost 1 označava da je taj odabir točan odgovor, dok 0 označava da odabir označava pogrešan odgovor

SIFRA – tekstualno polje, prima vrijednosti *a,b,c* ili *d*, služi za validaciju točnih odgovora na ispitu

U tablicu ODABIR spremaju se mogući odgovori na pitanje koje je određenom atributom PITANJE_ID.

ISPIT

ISPIT_ID – primarni ključ

KORISNIK_ID – vanjski ključ koji povezuje ispit s korisnikom

MAKSIMALNI_BROJ_BODOVA – numerička vrijednost maksimalnog broja bodova na ispitu

OSTVARENI_BROJ_BODOVA – numerička vrijednost ostvarenog broja bodova na ispitu

STATUS_ISPITA – tekstualna vrijednost statusa ispita, odnosno je li ispit položen ili nije

U tablicu ISPIT spremaju se ostvareni bodovi te status ispita koji je ostvario korisnik određen s atributom KORISNIK_ID.

ISPIT_PITANJE

ISPIT_PITANJE_ID – primarni ključ

ISPIT_ID – vanjski ključ na tablicu ispit

PITANJE_ID – vanjski ključ na tablicu pitanje

ODGOVOR – tekstualno polje koje označava korisnikov odgovor na pitanje

Tablica ISPIT_PITANJE je vezna tablica koja sadržava pitanja na ispitu određenom atributom ISIPT_ID, te odgovor na pitanje određenom atributom PITANJE_ID.

Tablice STATUS_POLAZNIKA, ULOGE i KATEGORIJE su šifrarničke tablice, odnosno tablice koje sadrže samo šifru i vrijednost nekog atributa, te ove tablice imaju sve zajedničke atribute osim primarnog ključa pa će se u nastavku opisati samo jedna od tih tablica. Analogno se sve primjenjuje na preostale tablice.

KATEGORIJE

KATEGORIJA_ID – primarni ključ

SIFRA – tekstualno polje koje označava šifru pod kojom je spremljena kategorija

NAZIV – tekstualno polje u koje se sprema vrijednost šifre

NAZIV_EN – tekstualno polje u koje se sprema engleska vrijednost šifre

KORISNIK

KORISNIK_ID – primarni ključ

ULOGA_ID – vanjski ključ na tablicu ULOGE, sprema se uloga korisnika

IME – tekstualno polje u koje se sprema ime korisnika

PREZIME – tekstualno polje u koje se sprema prezime korisnika

KORISNICKO_IME – tekstualno polje za korisničko ime potrebno za prijavu u sustav

EMAIL – tekstualno polje u koje se sprema e-mail, na koji korisnik može dobiti novu lozinku u slučaju da je zaboravio lozinku

LOZINKA – tekstualno polje u koju se sprema hashirana lozinka

OIB – tekstualno polje za spremanje osobnog identifikacijskog broja korisnika

U tablicu KORISNIK spremaju se svi registrirani korisnici sa svojim ulogama. Podaci iz tablice KORISNIK su povezani s tablicama INSTRUKTOR i POLAZNIK koje sadrže detaljnije podatke, ovisno o kojoj ulozi je riječ.

INSTRUKTOR

INSTRUKTOR_ID – primarni ključ

KORISNIK_ID – vanjski ključ na tablicu KORISNIK

BROJ_SLOBODNIH_MJESTA – numerička vrijednost koja označava koliko još polaznika može preuzeti instruktor

Tablica INSTRUKTOR osim podatka o korisniku sadrži i podatak koliko polaznika još može instruktor preuzeti.

VOZILO

VOZILO_ID - primarni ključ

KATEGORIJA_ID – vanjski ključ na tablicu KATEGORIJA

MARKA_VOZILA – tekstualna vrijednost koja označava marku vozila

MODEL – tekstualna vrijednost koja označava model vozila

REGISTRACIJA – tekstualna vrijednost koja prikazuje registracijske oznake vozila

U tablicu Vozilo spremaju se vozila koja ne sebe vežu određenu kategoriju preko atributa KATEGORIJA_ID.

INSTRUKTOR_VOZILO

INSTRUKTOR_VOZILO_ID – primarni ključ

INSTRUKTOR_ID – vanjski ključ na tablicu INSTRUKTOR

VOZILO_ID – vanjski ključ na tablicu VOZILO

INSTRUKTOR_VOZILO vezna je tablica koja omogućuje ostvariti vezu više na više između tablica INSTRUKTOR i VOZILO.

POLAZNIK

POLAZNIK_ID – primarni ključ

KORISNIK_ID – vanjski ključ na tablicu KORISNIK

INSTRUKTOR_ID – vanjski ključ na tablicu INSTRUKTOR

STATUS_POLAZNIKA_ID – vanjski ključ na tablicu STATUS_POLAZNIKA

ODABRANA_KATEGORIJA_ID – vanjski ključ na tablicu KATEGORIJE

Tablica POLAZNIK osim podatka o korisniku sadrži podatke o instruktoru, statusu polaznika i odabranoj kategoriji.

SAT_VOZNJE

SAT_VOZNJE_ID – primarni ključ

VOZILO_ID – vanjski ključ na tablicu VOZILO

BROJ_SATA – numerička vrijednost koji označava redni broj sata

OPIS – tekstualna vrijednost koja sadržava opis radnji rađenih na satu

DATUM – polje koje sadržava datum i vrijeme kada se održavao sat

U tablicu SAT_VOZNJE se spremaju satovi polaznika, a svi satovi jednog polaznika čine dnevnik vožnje.

SAT_POLAZNIK

SAT_POLAZNIK_ID – primarni ključ

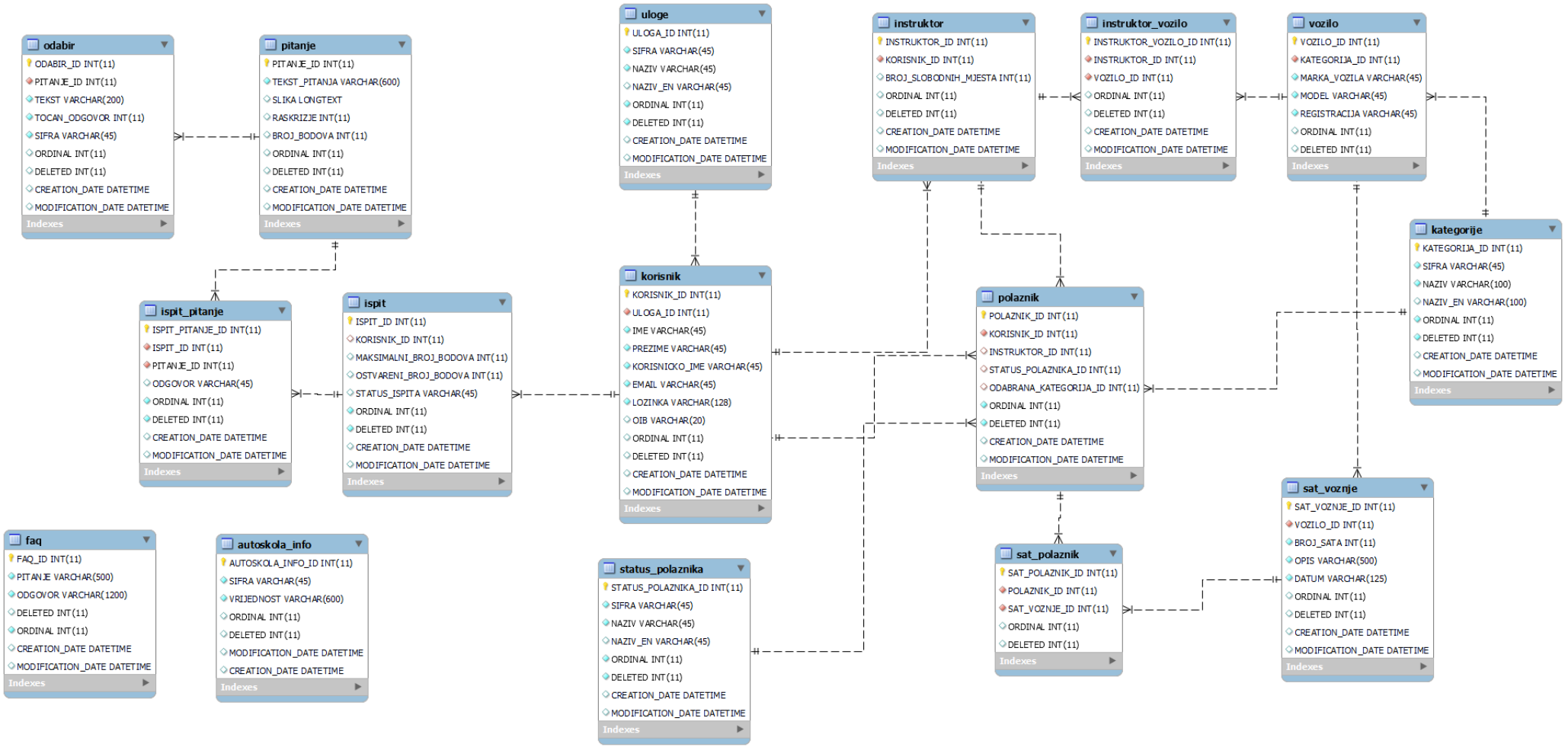
SAT_VOZNJE_ID – vanjski ključ na tablicu SAT_VOZNJE

POLAZIK_ID – vanjski ključ na tablicu POLAZNIK

Tablica SAT_POLAZIK je je vezna tablica koja omogućuje implementaciju veze više na više između tablica SAT_VOZNJE i POLAZIK.

Većina tablica još ime i sljedeće atribute:

- DELETED – vrijednost 0 označava da red nije izbrisan, a 1 da je izbrisan
- ORDINAL – poredak pojavljivanja redaka
- CREATION_DATE – datum i vrijeme kada je redak kreiran
- MODIFICATION_DATE – datum i vrijeme kada je redak modificiran.



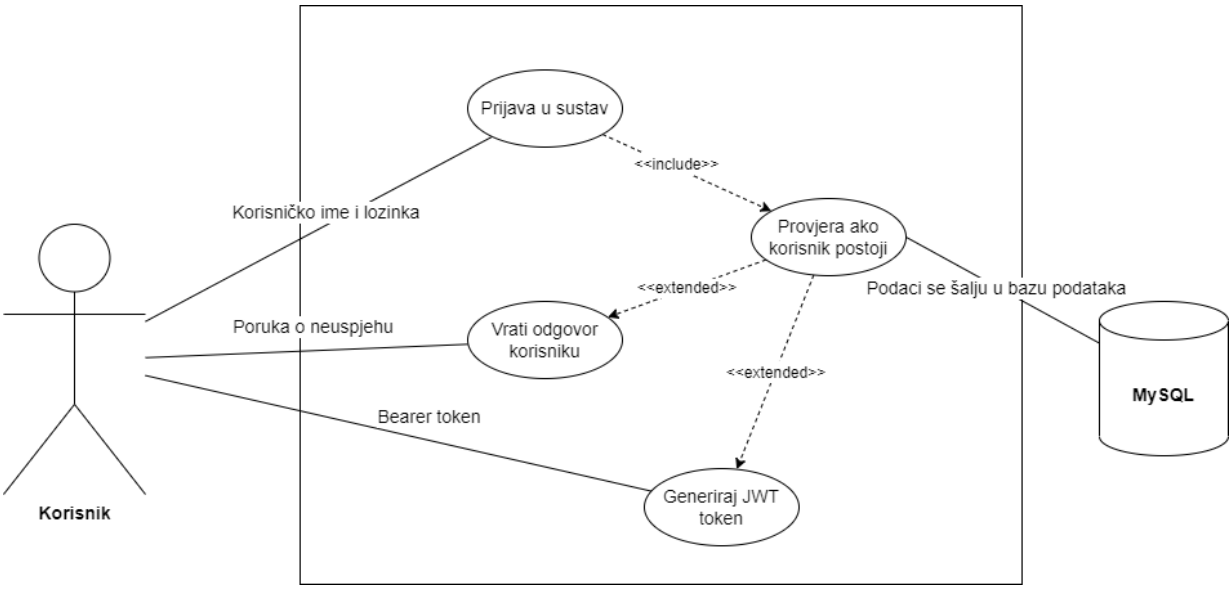
Slika 18. ERA dijagram aplikacije za vođenje autoškole [MySQL Workbrench]

9.3. Funkcionalnosti aplikacije za vođenje autoškole

U ovom poglavlju će biti prikazane glavne funkcionalnosti aplikacije za upravljanjem autoškole. Glavne funkcionalnosti su prijava i registracija u aplikaciju, rješavanje ispita iz propisa te vođenje dnevnika vožnje od strane instruktora. Za svaku funkcionalnost će biti prikazani zasloni korisničkog dijela, dijagrami slučajeva korištenja te bitni dijelovi kôda koji to omogućuju.

9.3.1. Prijava u aplikaciju za vođenje autoškole

Prijava u aplikaciju se vrši preko obrasca u koji korisnik treba unijeti korisničko ime i lozinku. Nakon što se unesu korisnički podaci, zahtjev se šalje na poslužiteljski dio gdje se obrađuje. Ovaj slučaj prikazan je na slici ispod u dijagramu slučajeva. Nakon što se preko korisničkog dijela na poslužiteljski pošalju podaci, korisničko ime i lozinka, kreće proces prijave u sustav. Na poslužiteljskom dijelu prvo se provjerava postoji li korisnik s navedenim korisničkim imenom u bazi te ukoliko postoji provjerava se da li je hashirana lozinka jednaka lozinki u bazi podataka. Ako nije, zahtjev neće biti uspješno obrađen te će se korisniku ispisati poruka o neuspješnoj prijavi. A u suprotnome, ako se podudaraju korisničko ime i lozinka, generira se JWT (*eng.* JSON Web Token) te se na korisnički dio šalje JWT token, podaci o prijavljenom korisniku te uloga korisnika.



Slika 19. Dijagram slučajeva za prijavu, pomoću Draw.io [Autorski rad]

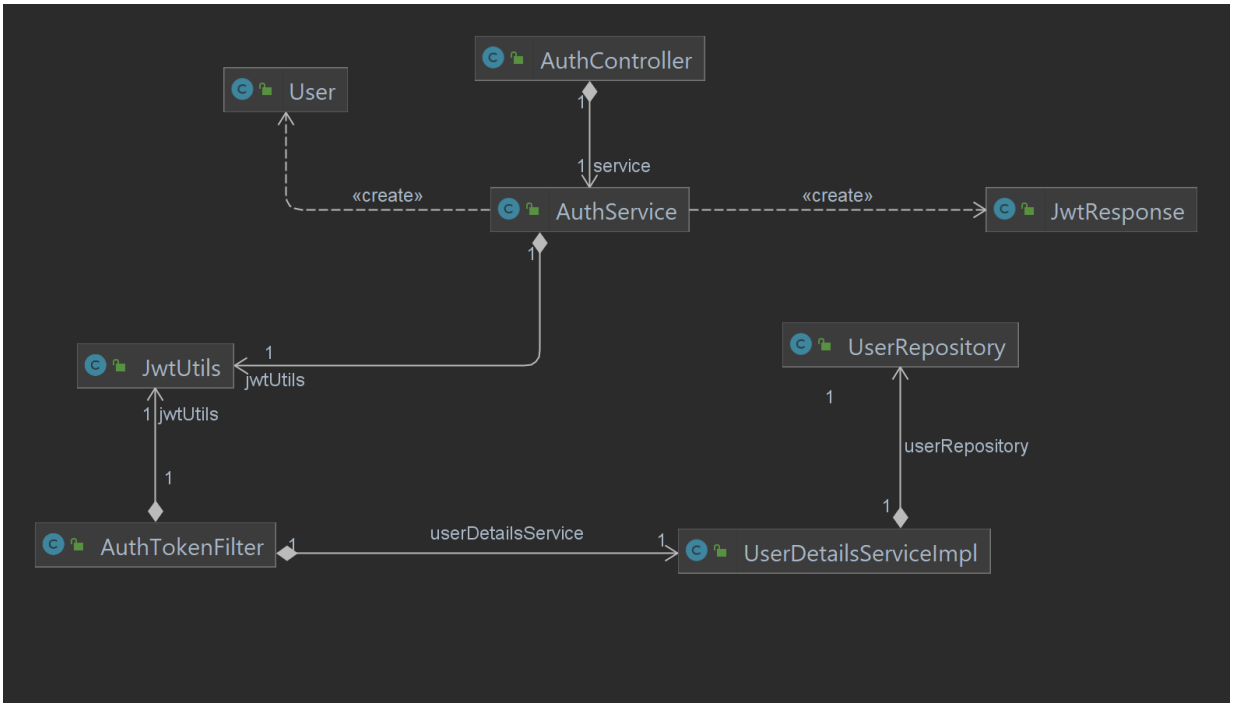
Na slici ispod prikazan je dijagram klasa za funkcionalnost prijave. Kao što se može primijetiti na slici, za implementaciju prijave korištene je 8 klasa. Sada će svaka klasa biti detaljnije opisana.

AuthController je klasa koja prima zahtjev od korisničkog dijela i od nje sve počinje. Njezin glavni zadatak je primiti zahtjev, proslijediti zahtjev u servis te vratiti odgovor. Ona sadrži instancu *AuthService* klase kojoj prosljeđuje sve zahtjeve.

AuthService je klasa u kojoj se događa sva logika prijave u sustav. Ona kreira instancu klase *User* i *JWTResponse*, te sadrži instancu klase *JWTUtils*. Pomoću tih klasa vrši se potrebna logika objašnjena kod dijagrama slučajeva korištenja, a sam kôd biti će kasnije prikazan.

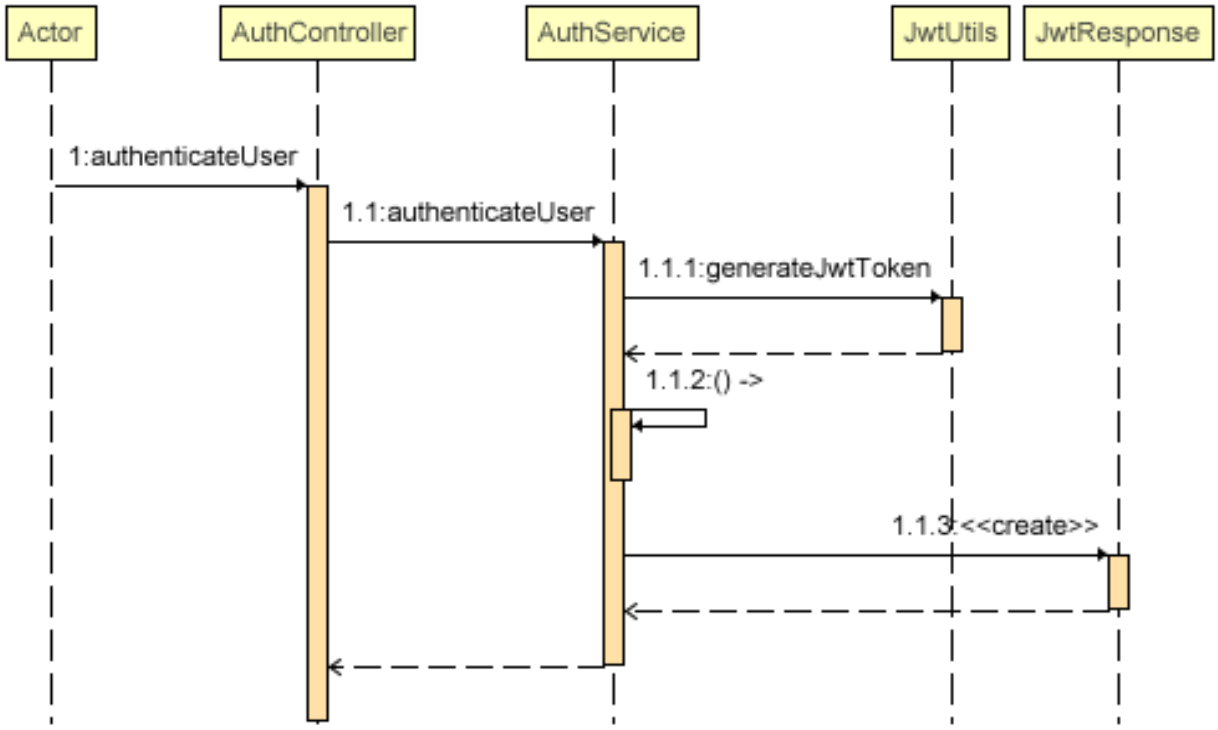
User i *JWTResponse* klase su jednostavne klase koje sadrže samo atribute. *User* klasa sadrži atribute tablice korisnik, kako bi se u nju mogli spremiti podaci o korisniku, ako postoji korisnik s danim korisničkim imenom i lozinkom, a klasa *JWTResponse* je klasa koja se vraća na korisnički dio te sadrži atribute poput uloge korisnika, imena i prezimena korisnika te atribut gdje je sadržan sam JWT token.

JWTUtils je klasa koja pomoću klasa *AuthTokenFIlter, UserDetailsServiceImpl* i *UserRepository* dohvaća korisnika, te ako postoji generira JWT token.



Slika 20. Dijagram klasa za prijavu, pomoću IntelliJ [Autorski rad]

Slijed izvođenja prijave prikazan je na dijagramu slijeda na slici ispod. Kao što se već prije spominjalo, slijed počinje kada korisnik aktivira metodu kontrolera za autentifikaciju. Kontroler prosljeđuje zahtjev na servis gdje pokreće istoimenu metodu. Servis zatim koristi instancu *JwtUtils* klase kako bi se generirao JWT token te se podaci spremaju u novo kreiran objekt tipa *JwtResponse* koji se šalje do servisa, a zatim prosljeđuje do kontrolera pa tako dolazi do korisničkog dijela.

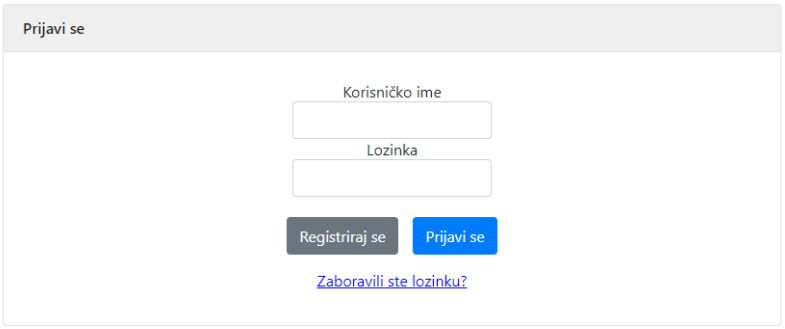


Slika 21. Dijagram slijeda za prijavu, pomoću IntelliJ [Autorski rad]

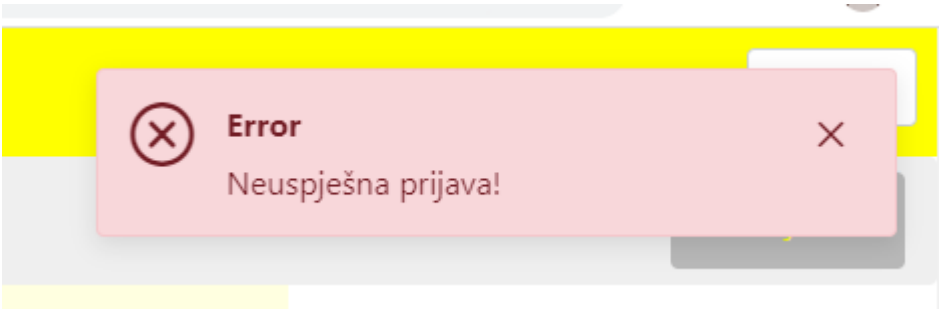
Sve klase su dosad opisane što rade, većina njih ima jednostavnu implementaciju budući da su zadužene za jednu stvar, no klasa *AuthService* odnosno njezina metoda *authenticateUser()* je metoda koja sve spaja i gdje se vrši sva logika. Njezin kôd prikazan je ispod ovog teksta. Dakle, u toj metodi prvo se preko *autehnticationManager* objekta prijavljuje korisnik te se sprema u kontekst. Ukoliko nema korisnika, metoda javlja grešku i korisnik dobije informaciju o neuspjeloj prijavi. Nakon toga se pomoću *jwtUtils* objekta generira JWT token za korisnika, koji ima svoje vrijeme isteka te se nakon tog vremena korisnik automatski odjavljuje. Kada se generirao token dohvaćaju se svi podaci o korisniku kao i njegova uloga koji se spremaju u novo kreirani *JwtResponse* objekt i šalju se natrag u kontroler.

```
public JwtResponse authenticateUser(@Valid @RequestBody LoginRequest
loginRequest) {

      Authentication authentication = authenticationManager.authenticate(
                        new
                  UsernamePasswordAuthenticationToken(loginRequest.getUsern
                  ame(), loginRequest.getPassword()));


      SecurityContextHolder.getContext().setAuthentication(authentication);
      String jwt = jwtUtils.generateJwtToken(authentication);

      UserDetailsImpl userDetails =
(UserDetailsImpl)authentication.getPrincipal();

      List<String> roles = userDetails.getAuthorities().stream()
                        .map(item -> item.getAuthority())
                        .collect(Collectors.toList());

      return new JwtResponse(jwt,
                              userDetails.getId(),
                              userDetails.getUsername(),
                              userDetails.getEmail(),
                              roles.get(0));
}
```

Nakon prezentacije implementacijskog dijela funkcionalnosti, vrijeme je da se prikaže kako korisnik to sve vidi na svojem zaslonu. Na slici 22 prikazan je obrazac za prijavu. Sastoji se od dva polja za unos, unos korisničkog imena i lozinke, dva gumba te teksta za zaboravljenu lozinku. Gumb *„Registriraj se“* vodi do obrasca za registraciju, tekst *„Zaboravili ste lozinku?“* otvara prozor za obnovu lozinke, dok gumb *„Prijavi se“* aktivira formu te šalje podatke upisane u poljima na poslužiteljski dio. Ukoliko su krivi korisnički podaci pojavljuje se poruka o neuspješnoj prijavi koja je prikazana na slici 23.



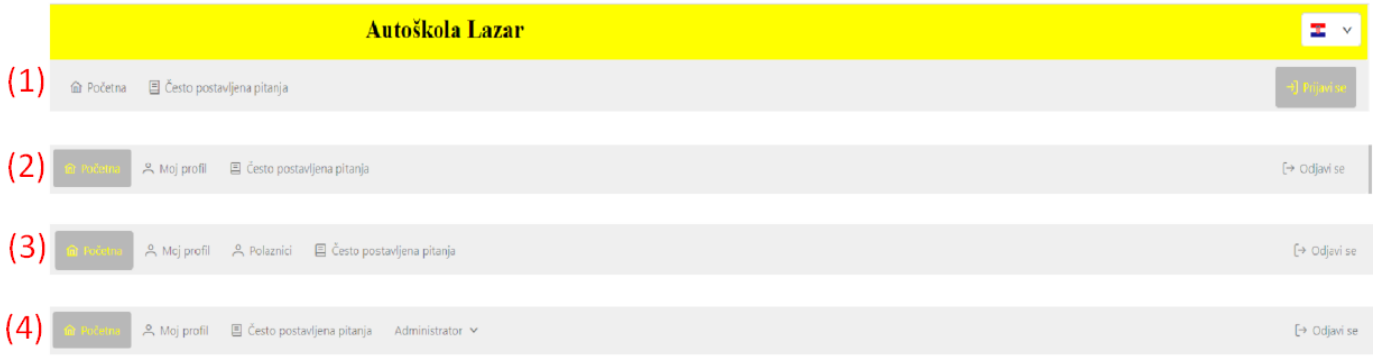
Slika 22. Prikaz obrasca za prijavu [Autorski rad]



Slika 23. Poruka o neuspješnoj prijavi [Autorski rad]

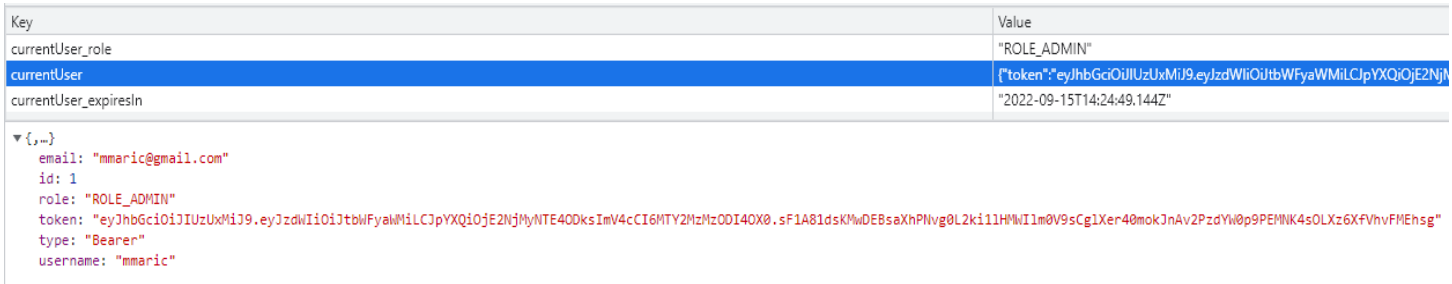
Ukoliko je prijava uspješna, prikaže se *Početna* stranica sa drugačijom navigacijskog trakom, ovisno o ulozi. Tako se na slici 24. mogu vidjeti četiri vrste navigacijske trake:

1. navigacijska traka za neregistrirane korisnike
2. navigacijska traka za polaznika autoškole (registrirani korisnik)
3. navigacijska traka za instruktore (moderatore)
4. navigacijska traka za administratora aplikacije



Slika 24. Navigacijske trake ovisno o ulozi korisnika [Autorski rad]

Nakon uspješne prije, osim što ste promijeni navigacijska traka, također su podaci o prijavi spremljeni u korisničkom dijelu u lokalnoj memoriji (*eng. Local storage*). Kao što se vidi na slici 25, podaci koji se spremaju su uloga, vrijeme isteka tokena te podaci o prijavljenom korisniku kao što su: e-mail, token te korisničko ime.



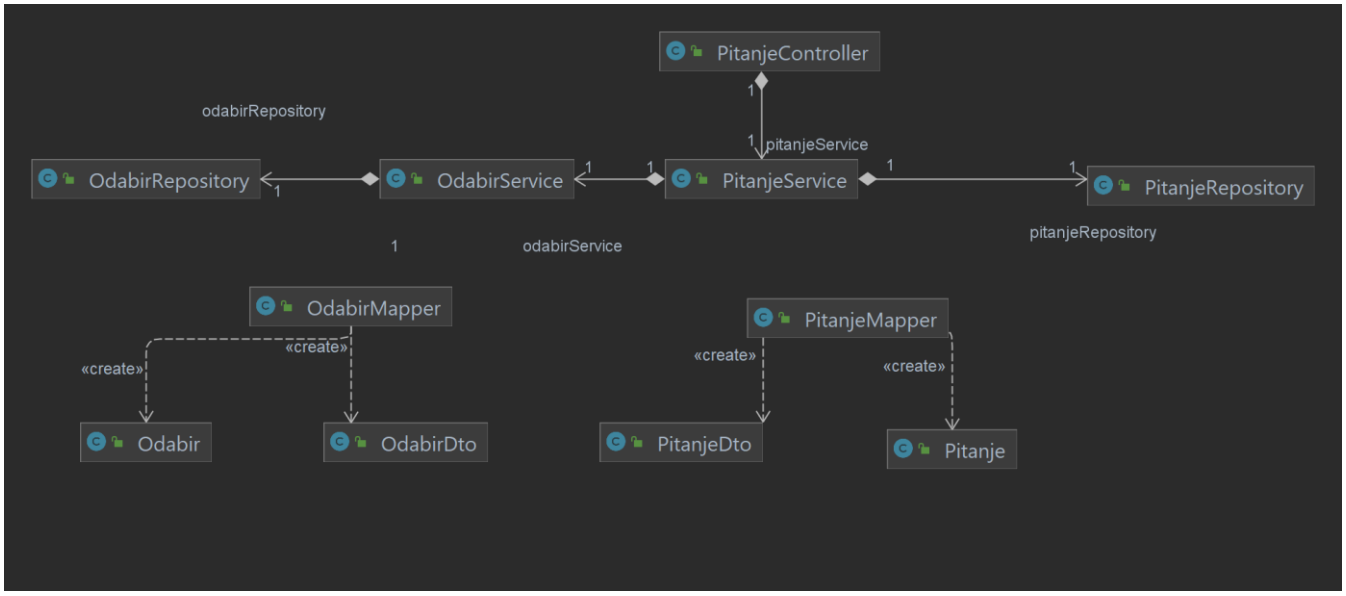
Slika 25. Spremljeni podaci na korisničkom dijelu nakon prijave [Autorski rad]

9.3.2. Generiranje i provjera ispita iz propisa

Sljedeća funkcionalnost je generiranje, rješavanje i provjera ispita iz propisa. Kada se polaznik registrira, ima opciju na stranici *Moj profil* upisati se u autoškolu tako da odabere kategoriju koju želi položiti. Nakon prijave u autoškolu, na istoj stranici se prikaže gumb za rješavanje ispita. Kada se klikne taj gumb, generira se ispit i počinje proces rješavanje ispita. Nakon što se ispit riješi, sprema se u bazu i spreman je za pregled. Ova funkcionalnost će se detaljnije objasniti kroz dijagram klasa, dijagram slijeda te uz same prikaze ispita iz aplikacije.

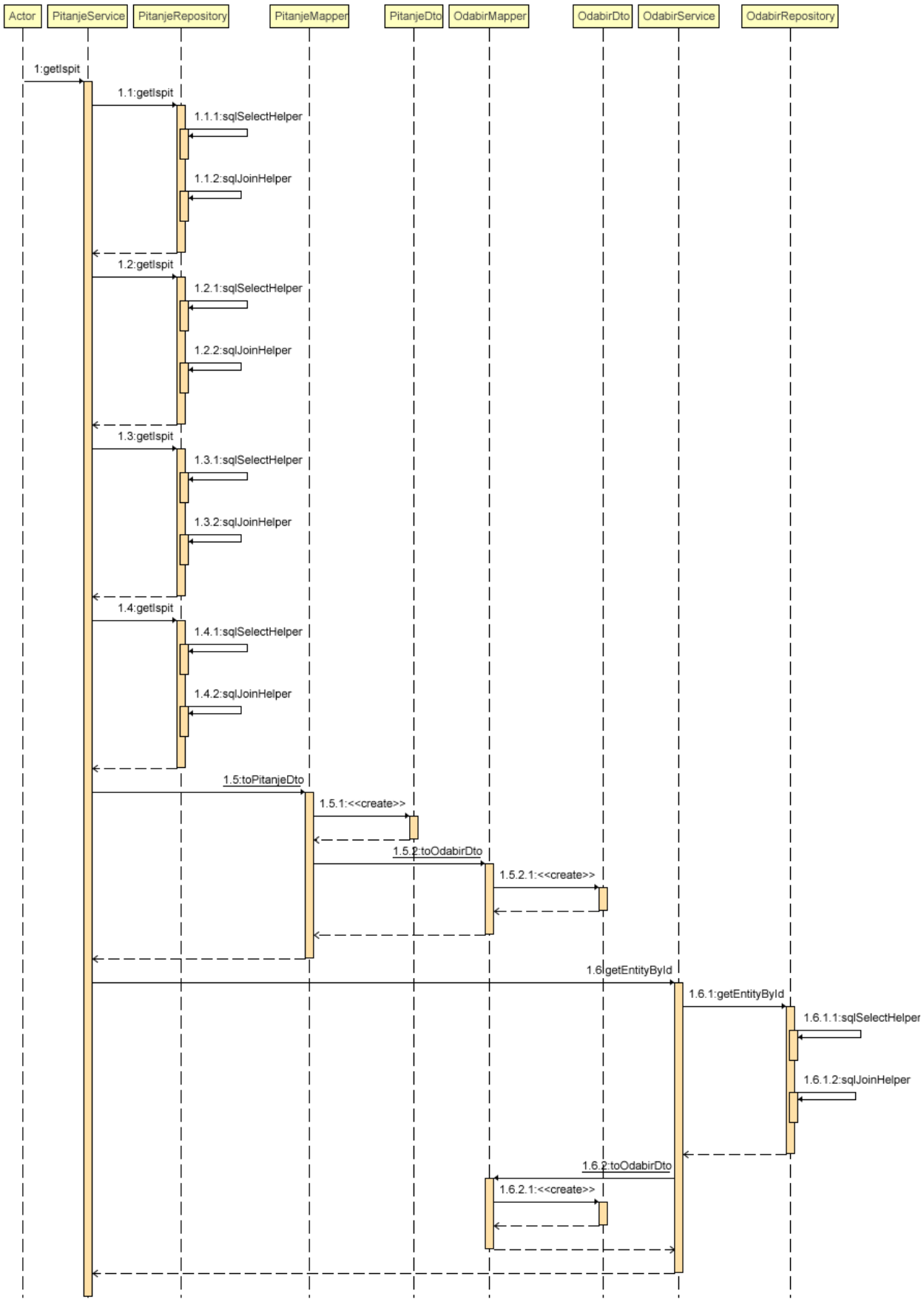
Na slici ispod prikazan je dijagram klasa za generiranje ispita. Budući da se ispit sastoji od više pitanja, za generiranje ispita koriste se klase pitanja kako bi se iz baze podataka dohvatila potrebna pitanja za ispit. Ovaj dijagram odgovora većini dijagrama u ovom projektu što se tiče operacija nad pojedinom entitetu. Svaki entitet ima ovih šest klasa koje služe za komuniciranje između korisničkog dijela i baze podataka preko poslužiteljskog dijela. Klase *Pitanje* i *PitanjeDto* su modelske klase, odnosno klase koje odgovaraju potrebnim modelima. Klasa *Pitanje* odgovara modelu pitanja iz baze podataka, a klasa *PitanjeDto* odgovara modelu podataka potrebnom na korisničkom dijelu da bi se podaci prikazali korisniku. Te dvije klase su povezane klasom *PitanjeMapper*. Ova klasa pretvara objekte *Pitanje* u *PitanjeDto* i obratno. Ove tri klase su potrebne zbog toga što su odvojeni pogledi na entitete, odnosno posebni pogled na entitete ima korisnički dio, a poseban pogled ima baza podataka. Komunikacija između poslužiteljskog dijela i baze podataka koristi klasu *Pitanje*, dok komunikacija između poslužiteljskog i korisničkog dijela koristi klasu *PitanjeDto*. Klasa *PitanjeController* je klasa koja komunicira sa poslužiteljskim dijelom te ona koristi samo klasu *PitanjeDto*, dok je *PitanjeRepository* klasa koja komunicira samo sa bazom podataka pa ona koristi klasu *Pitanje*. U klasi *PitanjeService* događa se sva potrebna poslovna logika te ona ima pristup i jednom i drugom modelu, tj. *PitanjeController* klasa šalje *PitanjeDto* klasu u servisnu klasu, tamo se ti podaci obrađuju, a zatim se pomoću *PitanjeMappera* pretvaraju u klasu *Pitanje* kako bi se ti podaci mogli spremiti u bazu podataka.

U konkretnom primjeru za generiranje ispita kada korisnik klikne gumb *„Započni test“*, klasa *PitanjeController* zaprimi zahtjev za generiranje testa te šalje taj zahtjev prema klasi *PitanejService* koja po zadanim kriterijima ispita iz propisa generira pitanja za taj ispit tako da šalje zahtjeve prema repozitoriju. Kad se dohvate sva pitanja, budući da je *Pitanje* kompleksniji entitet te sadrži u sebi listu *Odabira*, za svako pitanje se još moraju dohvatiti odabiri, a odabiri se dohvaćaju iz klase *OdabirService* koja pristupa repozitoriju klase *Odabir* kao bi dohvatila sve odabire za pojedinačno pitanje iz baze podataka. Jednom kada se svi podaci dohvate, lista pitanje šalje se u *PitanjeController*, a iz kontrolera do korisničkog dijela aplikacije.



Slika 26. Dijagram klasa za generiranje ispita, pomoću IntelliJ [Autorski rad]

Na slici ispod nalazi se dijagram slijeda za generiranje ispita. Već prije je objašnjeno kako se dolazi do klase *PiatnjeService* i metode za generiranje ispita *getIspit()*. Metoda *getIspit()* u servisnoj klasi četiri put poziva istoimenu metodu repozitorija. Razlog tome bit će objašnjen kasnije kada će biti prikazan kôd te servisne metode. Nakon što se dohvate sva pitanja za ispit, potrebno ih je konvertirati iz klase *Pitanje* u klasu *PitanjeDto* kako bi bile u modelu potrebnom za prikaz na korisničkom dijelu aplikacije. Kada su se sva pitanja prebacila u klasu *PitanjeDto* potrebno je na svako pitanje dodati listu odabira, tj. ponuđenih odgovora. Kroz petlju se prolazi kroz sva pitanja te se pomoću servisne klase odabira dobe svi odabiri za pojedinačno pitanje. Kada se to završi, lista pitanje se šalje prema kontroleru pa prema korisničkom dijelu.



Slika 27. Dijagram slijeda za generiranje ispita, pomoću IntelliJ [Autorski rad]

U kôdu ispod, prikazane su metode iz klase *PitanjeService* i *PitanjeRepository* za generiranje ispita . Prva metoda je iz klase *PitanjeService*, dok je druga iz *PitanjeRepository*. Kada kontroler pozove servisnu metodu *getIspit()*, četiri put se poziva metoda iz repozitorija za dohvaćanje pitanja. U metodi *getIspit()* iz repozitorija može se primijetiti da se na kraju upita postavlja kriterij za broj bodova koje pitanje nosi i koliko pitanja će se dohvatiti sa tim brojem bodova. Dio upita *„order by rand()“* označava da se dohvaćaju pitanja po nasumičnom odabiru tako da se svaki put dohvate različita pitanja iz te kategorije pitanja. Metoda *getIspit()* iz repozitorija se poziva četiri puta, ali sa drugim varijablama zbog toga što ispit iz propisa uvijek ima maksimalni broj bodova 120, te ima četiri pitanja sa sedam bodova, odnosno četiri raskrižja. Ostala pitanja i broj bodova su složeni tako da ukupan broj bodova bude 120 i da uvijek bude 38 pitanja. Kada se dohvati svih 38 pitanja, pomoću klase *PitanjeMapper* se konvertiraju u listu s klasom *PitanjeDto*. Zatim se prolazi *for* petljom kroz svih 38 pitanja te se svakom pitanju preko *OdabirService*a dodaju odabiri za to pitanje. Kada se završi s time, korisničkom dijelu se vraća lista pitanja za ispit.

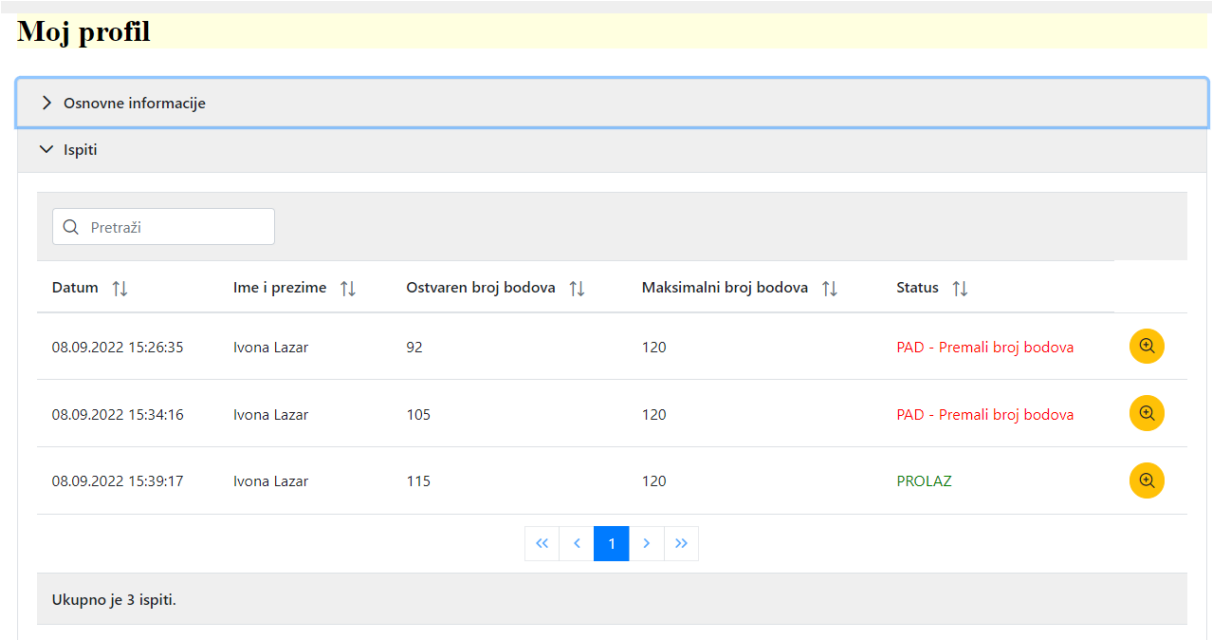
```
public List<PitanjeDto> getIspit(){

	List<Pitanje> modelEntities=  pitanjeRepository.getIspit(7, 4);
	modelEntities.addAll(pitanjeRepository.getIspit(4, 6));
	modelEntities.addAll(pitanjeRepository.getIspit(3, 12));
	modelEntities.addAll(pitanjeRepository.getIspit(2, 16));

	List<PitanjeDto> response =
	modelEntities.stream().map(PitanjeMapper::toPitanjeDto).collect(Colle
	ctors.toList());

	if(response != null) {
		for(PitanjeDto pitanjeDto : response) {

	pitanjeDto.setOdabiri(odabirService.getEntityById(pitanjeDto.getPitan
jeId().toString()));
		}
	}

	return response;
}

public List<Pitanje> getIspit(int numberOfPoints, int numberOfQuestions) {
	StringBuilder query = new StringBuilder();
	MapSqlParameterSource parameters = new MapSqlParameterSource();

	sqlSelectHelper(query);
	sqlJoinHelper(query);
	query.append(" AND " + MAIN_TABLE + "." + BROJ_BODOVA + " = :brojBodova
order by rand() limit :brojPitanja");

	parameters.addValue("brojBodova", numberOfPoints);
	parameters.addValue("brojPitanja", numberOfQuestions);

	return njdbc.query(query.toString(), parameters,
BeanPropertyRowMapper.newInstance(Pitanje.class)); }
```

Na slici ispod je prikazan kako izgleda dio ispita iz propisa koji se dobije kada se klikne gumb *„Započni test“*. Na slici su prikazana samo četiri pitanja, ali kao što se prije reklo, test se sastoji od 38 pitanja. Pitanja su većinom višestrukog odabira, gdje može biti točan jedan ili više ponuđenih odgovora, ali ima i pitanja gdje se traži od polaznika da upiše numeričku vrijednost. Ispit se može završiti klikom na gumb *„Završi test“* koji se nalazi na dnu testa, ali isto tako se može završiti izlaskom iz stranice. Dakle, kada se klikne spomenuti gumb ili kada se želi promijeniti stranica usred ispita, polazniku se pojavi prozorčić koji ga još jednom pita ako je siguran da želi završiti ispit. Ako klikne ne, ostane dalje na toj stranici rješavajući ispit, a ako klikne da, onda napušta tu stranicu te mu se spreme odgovori na pitanja na koje je do tog trenutka odgovorio.

Slika 28. Prikaz ispita iz propisa [Autorski rad]

Nakon što je ispit završio i ako ispit nije položen, polaznik još uvijek može dalje polagati ispit tak dugo do kad ga ne položi. Ispit je položen ako polaznik skupi 90% bodova, odnosno 108 bodova od 120, a pri tome mora odgovoriti točno na sva četiri pitanja vezana uz raskrižja. Kad položi ispit, više ne može pristupiti novom ispitu te status mu se mijenja iz *„Upisan“* u *„Položio ispit“*. Kada polaznik pristupi jednom ispitu, na stranici *Moj profil* pod

karticom *Ispiti* može vidjeti rezultate svih svoji pokušaja prikazanih tablično. Kao što je prikazano na slici ispod.



Slika 29. Prikaz ispita jednog polaznika [Autorski rad]

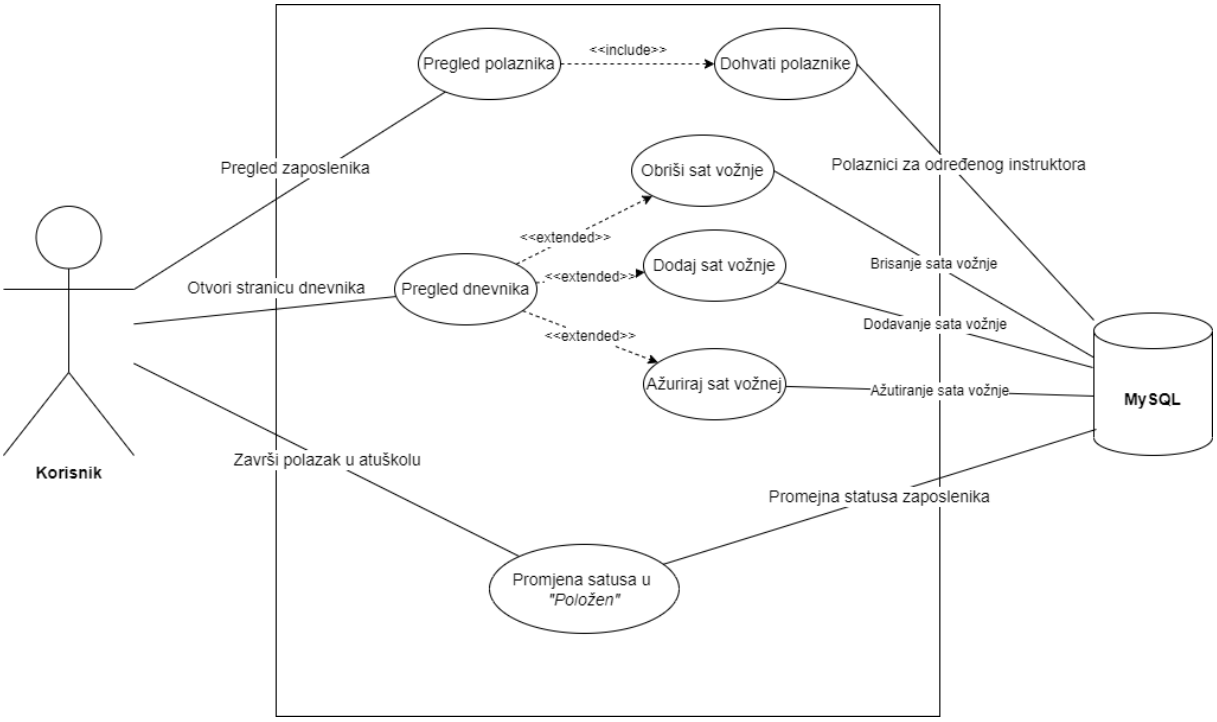
U zadnjem stupcu nalazi se gumb sa povećalom. Klikom na taj gumb otvara se ispit te su prikazane detaljne informacije o ispitu, kao i svih 38 pitanja koja su ispravljena. Dakle, za svako pitanje u donjem desnom kutu piše broj bodova. Ako je točno odgovoreno na pitanje onda se dobiva maksimalni broj bodova za to pitanje, a ako je krivo odgovoreno dobije se nula bodova. Za točan odgovor potrebno je imati odabrano sve točne ponuđene odgovore te se ne smije odabrati odabir koji nije točan. Naravno, kod pitanja gdje je trebalo upisati vrijednost, ta vrijednost mora biti jednaka točnom odgovoru kako bi se dobio maksimalni broj bodova. Kod pitanja višestrukog odabira, svi odabiri koji su točni označeni su zelenom bojom, dok su krivo odgovoreni označeni crvenom bojom. Primjer pregleda ispita može se vidjeti na slici ispod.

Slika 30. Pregled ispita [Autorski rad]

9.3.3. Vođenje dnevnika od strane instruktora

Još jedna od važnijih aktivnosti je vođenje dnevnika od strane instruktora. Instruktor ima pristupa stranici *Polaznici* gdje ima popis svih polaznika koji su njega odabrali za instruktora. Na ovaj stranici on može pristupiti dnevniku vožnje svih svojih polaznika te može uređivati isti. Uređivanje dnevnika znači da ima pravo za čitanje, kreiranje, ažuriranje i brisanje zapisa dnevnika, odnosno ima pravo na sve CRUD (*eng.* Create Read Update Delete) operacije.

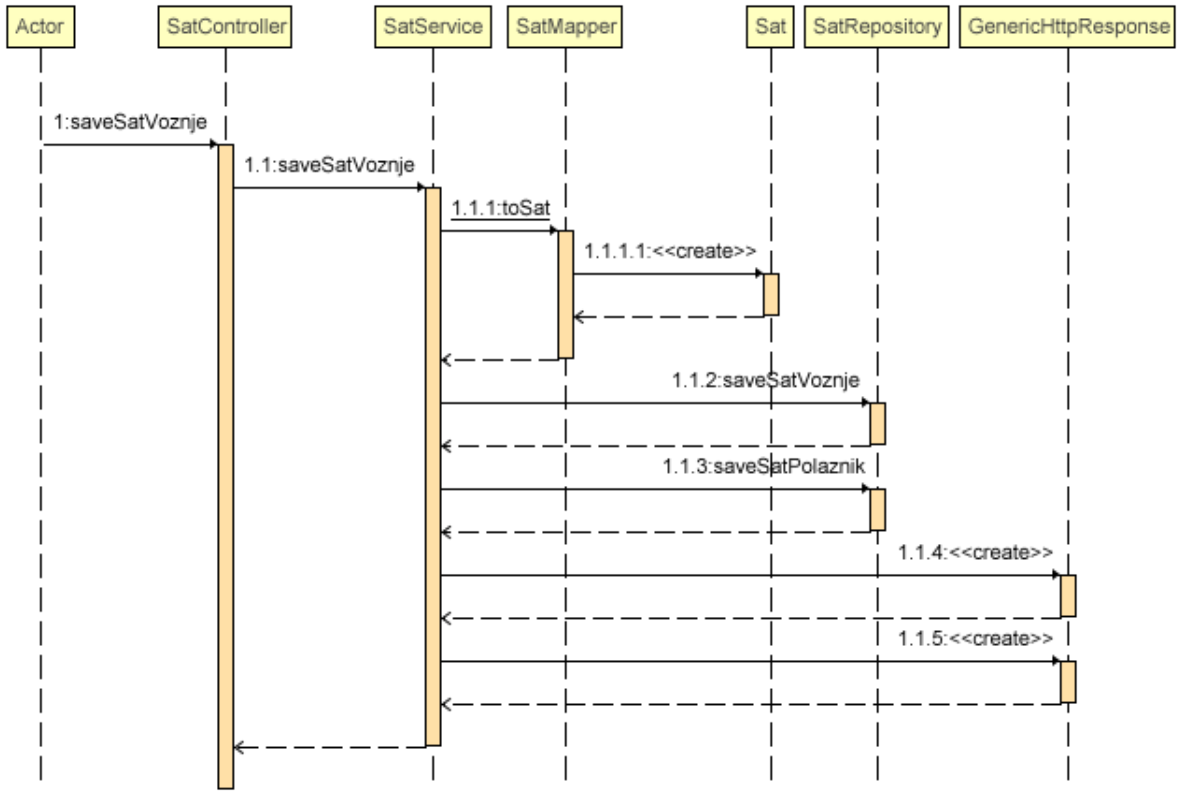
Na slici dolje prikazan je cijeli dijagram slučajeva navedene funkcionalnosti. Kao što je ranije spomenuto, instruktor ima pogled na sve svoje polaznike koji se dohvaćaju iz baze podataka. Za svakog polaznika ima opciju da pregleda njegov dnevnik. Dnevnik se zapravo sastoji od liste satova vožnje pa tako instruktor osim što može vidjeti sve satove vožnje, on ih može dodavati, uređivati i brisati. Svaka akcija se sprema u bazi podataka. Također, kada polaznik skupi minimalni broj satova vožnje, instruktor ima opciju da polazniku promijeni status u *„Završen“* što znači da je polaznik prošao ispit i vožnje te da je završio s autoškolom.



Slika 31. Dijagram slučajeva za vođenje dnevnika od strane instruktora, pomoću Draw.io [Autorski rad]

Budući da se u prošloj aktivnosti objasnilo kako se dohvaćaju podaci iz baze, kod ove aktivnosti će se detaljnije objasniti kako se rade osnovne operacije dodavanja, brisanja i ažuriranja podataka. Dijagram klasa se može analogno povezati s dijagramom klasa za generiranje ispita pa se ovdje neće prikazivati dijagram klasa.

Na slici ispod prikazan je dijagram slijeda. Iz dijagram slijeda izbačene su klase i metode koje samo konvertiraju vrijednosti iz klase za korisnički dio u klasu koju poznaje baza podataka. Na primjeru iz slike radi se o spremanju sata vožnje te isto kao i prije, prvo zahtjev dođe do kontrolera koji prosljeđuje zahtjev servisnoj klasi zaduženoj za taj zahtjev. Servisna klasa prvo pretvara objekt namijenjenog za korisnički dio u objekt namijenjen za bazu podataka. Nakon toga šalje podatke u klasu repozitorija određenog entiteta koja taj objekt sprema u bazu. U ovom slučaju dva put se poziva klasa repozitorija, zbog toga što se prvo trebaju podaci spremiti u tablicu SAT_VOZNJE, a zatim u tablicu SAT_POLAZNIK. Ukoliko je došlo do promijene u bazi, odnosno ako se dodao redak u tablicu, vraća se poruka o uspješnom dodavanju retka, a u suprotnom poruka da nije ništa dodano.



Slika 32. Dijagram slijeda za vođenje dnevnika od strane instruktora, pomoću IntelliJ [Autorski rad]

Ispod je prikazan kôd *SatController* klase. Može se primijetiti da sama klasa iznad sebe ima dvije anotacije, *@RestController* i *@RequiredArgsConstructor*. *@RequiredArgsContructor* generira konstruktor s jednim parametrom za svako polje. Sva neinicijalizirana konačna polja dobivaju parametar. *@RestController* je spoj dviju anotacija, *@Controller* i *@RequestBody*. Ova anotacija označava da je klasa rukovoditelj zahtjeva, odnosno da prima zahtjeve od strane klijenta ili drugih servisa. Nakon toga mogu se primijetiti *@GetMapping*, *@PostMapping* i *@PutMapping* anotacije. *@GetMapping* se koristi za dohvaćanje nekog entiteta, *@PostMapping* za unos entiteta u bazu te *@PutMapping* za ažuriranje objekta u bazi. U ovom slučaju za brisanje entiteta se ne koristi *@DeleteMapping* zbog toga jer se u bazi vrši brisanje na način da se ažurira atribut DELETED u tablici pa se zbog toga koristi *@PutMapping*. Za metodu dohvata satova vožnje, potrebno je proslijedit podatak o ID-u polaznika, dok kod ostalih metoda, potrebno je proslijediti cijeli objekt *SatDto*.

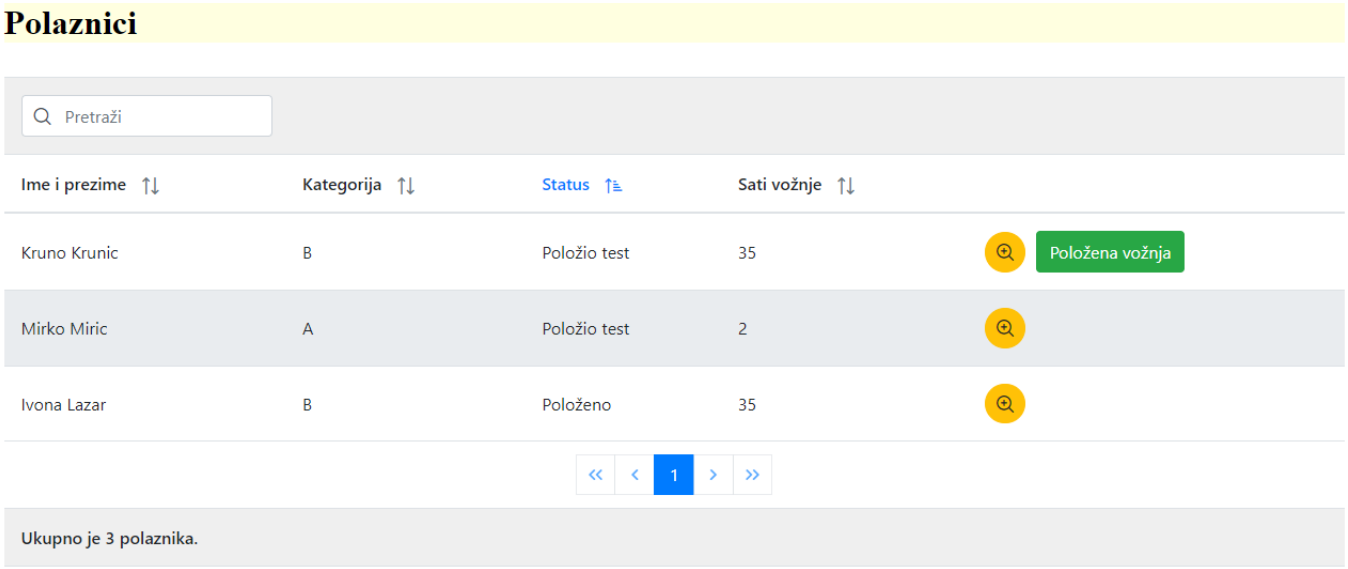
```
@RestController
@RequiredArgsConstructor
public class SatController {

	private final SatService satService;

	@GetMapping(value = "/api/satoviPolaznika")
	public ResponseEntity<List<SatDto>>
getEntityByKorisnikId(@RequestParam("id") String id) {
		List<SatDto> satDto = satService.getEntitiesByPolaznikId(id);
		return new ResponseEntity<>(satDto, HttpStatus.OK);
	}

	@PostMapping(value="/api/insertSatVoznje")
	public GenericHttpResponse<Long> saveSatVoznje(@RequestBody SatDto
satDto) {
		return satService.saveSatVoznje(satDto);
	}

	@PutMapping(value="/api/updateSatVoznje")
	public GenericHttpResponse<Long> updateSatVoznje(@RequestBody SatDto
satDto) {
		return satService.updateSatVoznje(satDto);
	}

	@PutMapping(value="/api/deleteSatVoznje")
	public GenericHttpResponse<Long> deleteSatVoznje(@RequestBody SatDto
satDto) {
		return satService.deleteSatVoznje(satDto);
	}

}
```

Ovdje ispod je prikaz kôd servisne klase za dodavanje sata vožnje. Servisna klasa ima anotaciju *@Service*, a ona označava klasu koja pruža poslovnu logiku te Spring kontekst automatski detektira ovu klasu kada se koristi konfiguracija bazirana na anotacijama i skeniranju staze klase (*eng. Classpath)*. Sama metoda za spremanje sata vožnje ima anotaciju *@Transactional* koja se koristi zbog spremanja podataka u više tablica u bazi. Ta anotacija služi tome da ako se dogodi greška prilikom promijene podataka u jednoj tablici, da se onda ne radi ni promjena podataka u drugoj tablici.

Logika servisne metode za unos sata vožnje je sljedeća. Kontroler proslijedi klasu koja se koristi na korisničkom dijelu pa ju servisna metoda prvo mora konvertirati u klasu poznatoj bazi podataka. Nakon toga se podaci spremaju u tablicu SAT_VOZNJE metodom saveSatVotnje(). Ako metoda vrati vrijednost veću od nula, odnosno ako se promijenio redak u tablici, onda se spremaju podaci i u tablicu SAT_POLAZNIK. Ako je sve prošlo u redu, vraća se odgovor korisniku da je entitet unesen, a u suprotnom vraća da ništa nije uneseno.

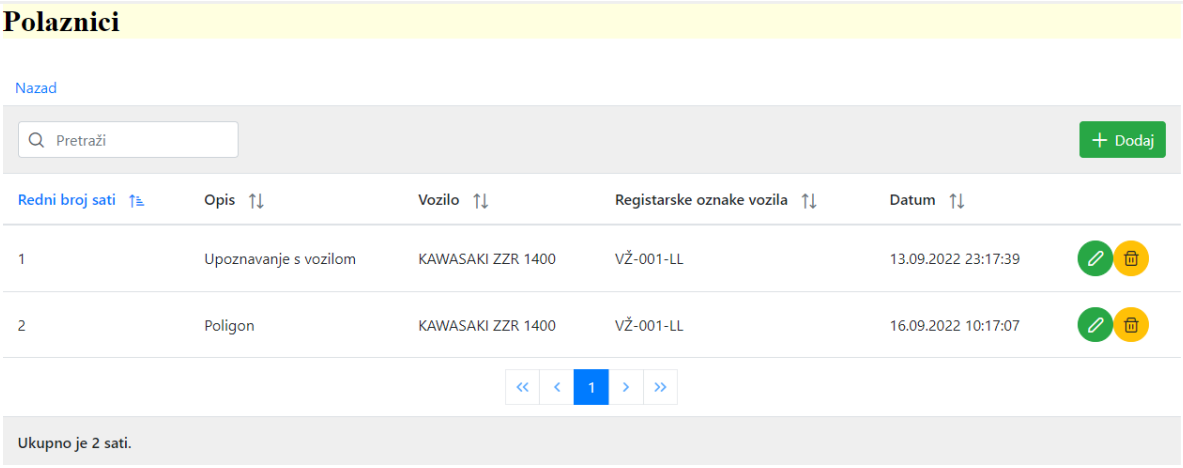
```
@Service
@RequiredArgsConstructor
public class SatService {

	@Transactional
	public GenericHttpResponse<Long> saveSatVoznje(SatDto satDto) {
		GenericHttpResponse<Long> response;

		Long id = satRepository.saveSatVoznje(SatMapper.toSat(satDto));
		if(id > 0) {

			satRepository.saveSatPolaznik(id.toString(),
satDto.getPolaznikId().toString());

			response = new
GenericHttpResponse<>(ResponseMessageEnum.ENTITY_INSERTED);

			response.setData(id);
		}else {
			response = new
GenericHttpResponse<>(ResponseMessageEnum.NOTHING_INSERTED);
		}
		return response;
	}
}
```

Instruktor na stranici *Polaznici* ima tablični prikaz svih svoji polaznika kao što je vidljivo na slici 33. Ovdje instruktor ima informacije u imenu i prezimenu polaznika, o kategoriji koju pohađa, njegovom statusu te broju odvoženih sati. Ukoliko korisnik ima minimalno 35 odvoženih sati, instruktoru se javlja opcija *„Položena vožnja“*. Ako se odabere ova opcija, to znači da je korisnik položio ispit iz vožnje te da je završio s autoškolom te mu se promijeni status u *„Položeno“*. Također instruktor može kliknuti gumb sa povećalom koji mu onda otvara dnevnik vožnje polaznika.

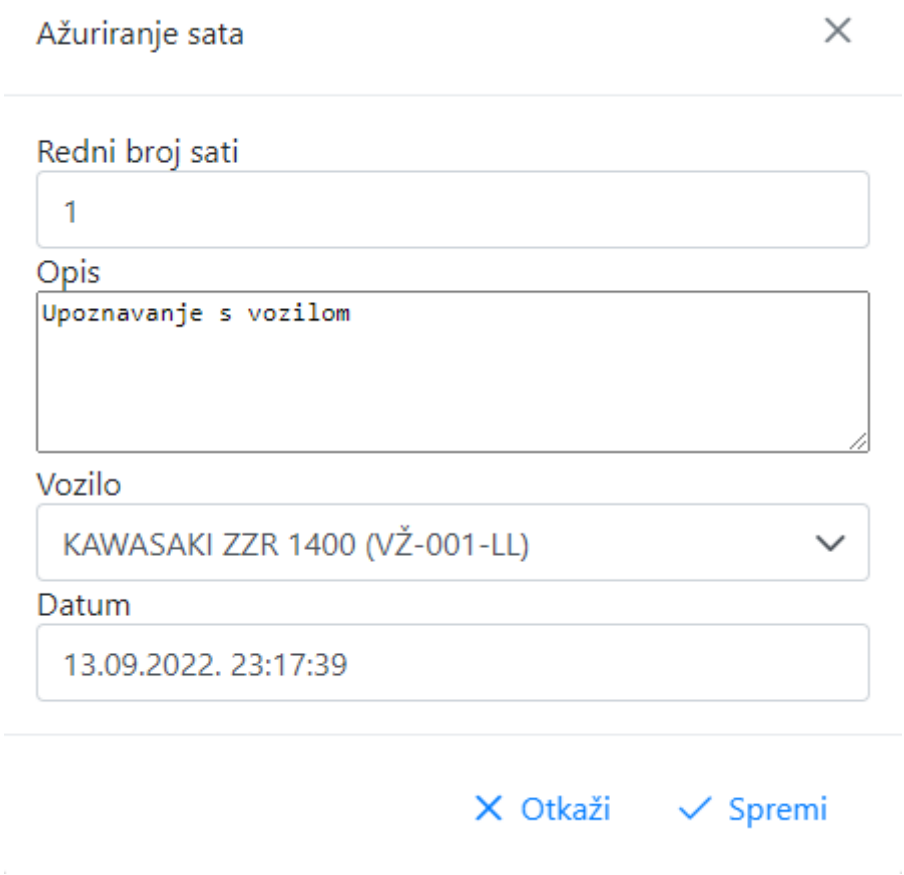


Slika 33. Prikaz polaznika jednog instruktora [Autorski rad]

Dnevnik vožnje polaznika prikazan je na slici 34. U ovom tabličnom prikazu instruktor vidi sve satove vožnje jednog polaznika. Prikazane informacije o satu su sljedeće: redni broj sata, opis, vozilo, registracija vozila te datum kada se polagao sat. Instruktor ovdje može dodavati, ažurirati te brisati satove. Forma za dodavanje ili ažuriranje prikazana je na slici 35.



Slika 34. Prikaz dnevnika vožnje [Autorski rad]



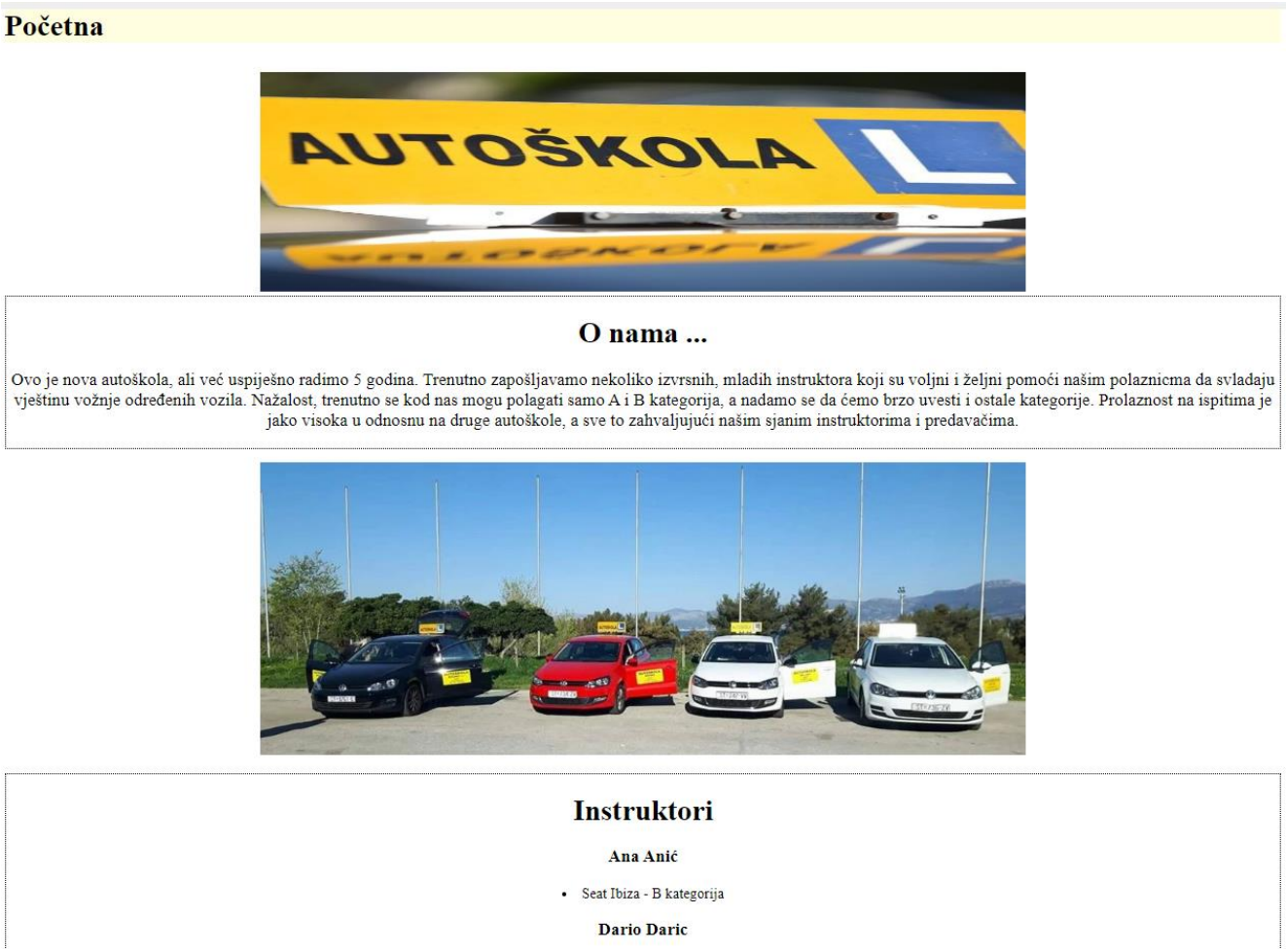
Slika 35. Prikaz forme za ažuriranje sata [Autorski rad]

9.3.4. Ostale stranice aplikacije

Ovdje će se prikazati ostale stranice aplikacije, ali bez implementacije već samo izgled i svrha stranica.

Početna stranica

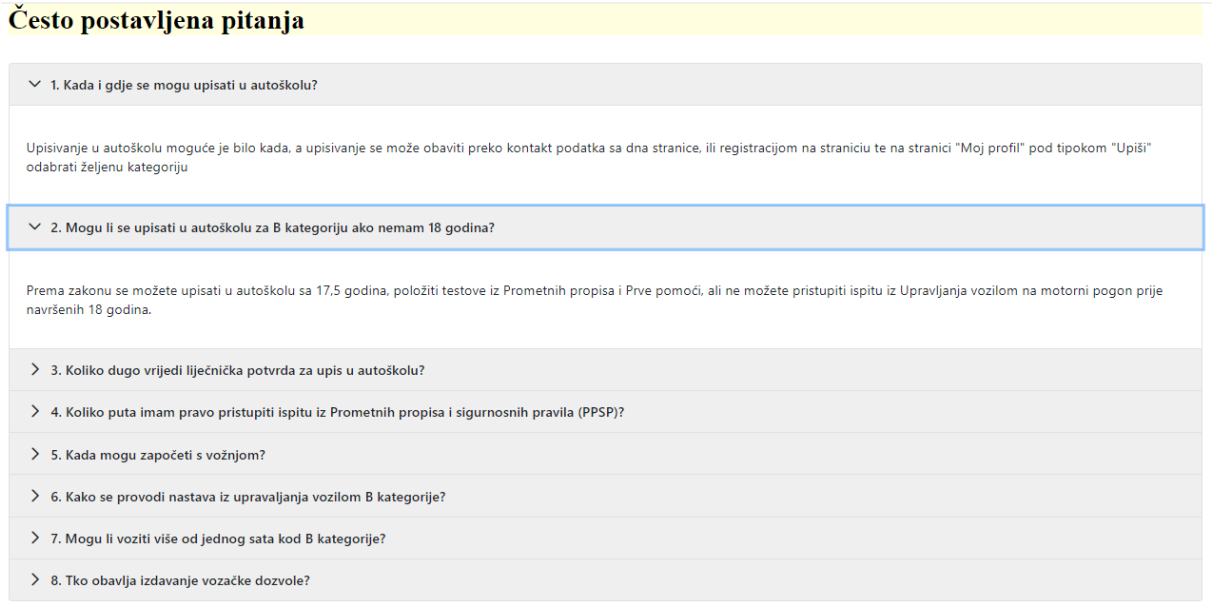
Početna stranica je prikazana na slici 36. Na njoj se nalazi kratak tekst o autoškoli koji administrator može mijenjati preko aplikacije. Također se nalazi i popis instruktora te kojim vozilima ti instruktorima upravljaju.



Slika 36. Prikaz početne stranice [Autorski rad]

Često postavljena pitanja

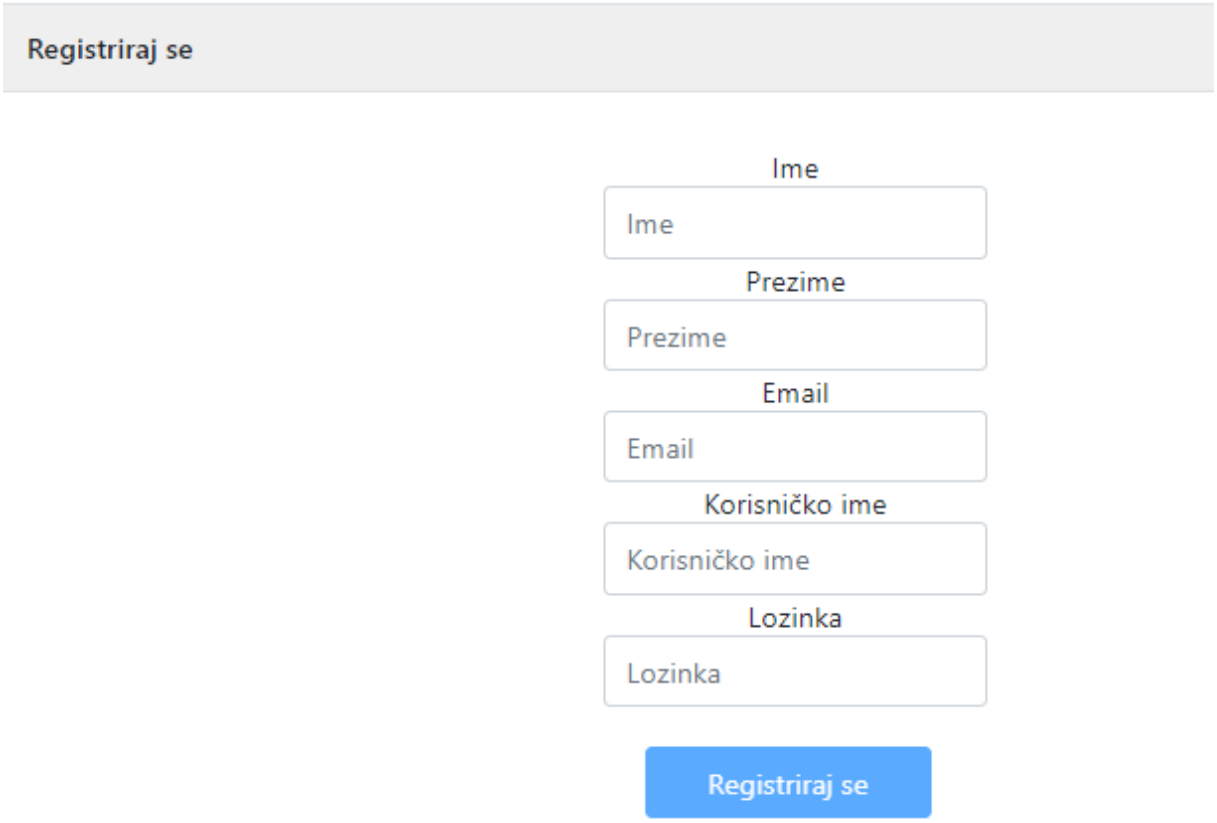
Prikaz stranice *Često postavljena pitanja* se nalazi na slici 37. Na ovoj stranici moguće je vidjeti odgovore na često postavljena pitanja, tako da se mogu dobiti neki odgovori bez dodatnog kontaktiranja autoškole.



Slika 37. Prikaz stranice sa čestim postavljenim pitanjima [Autorski rad]

Obrazac za registraciju

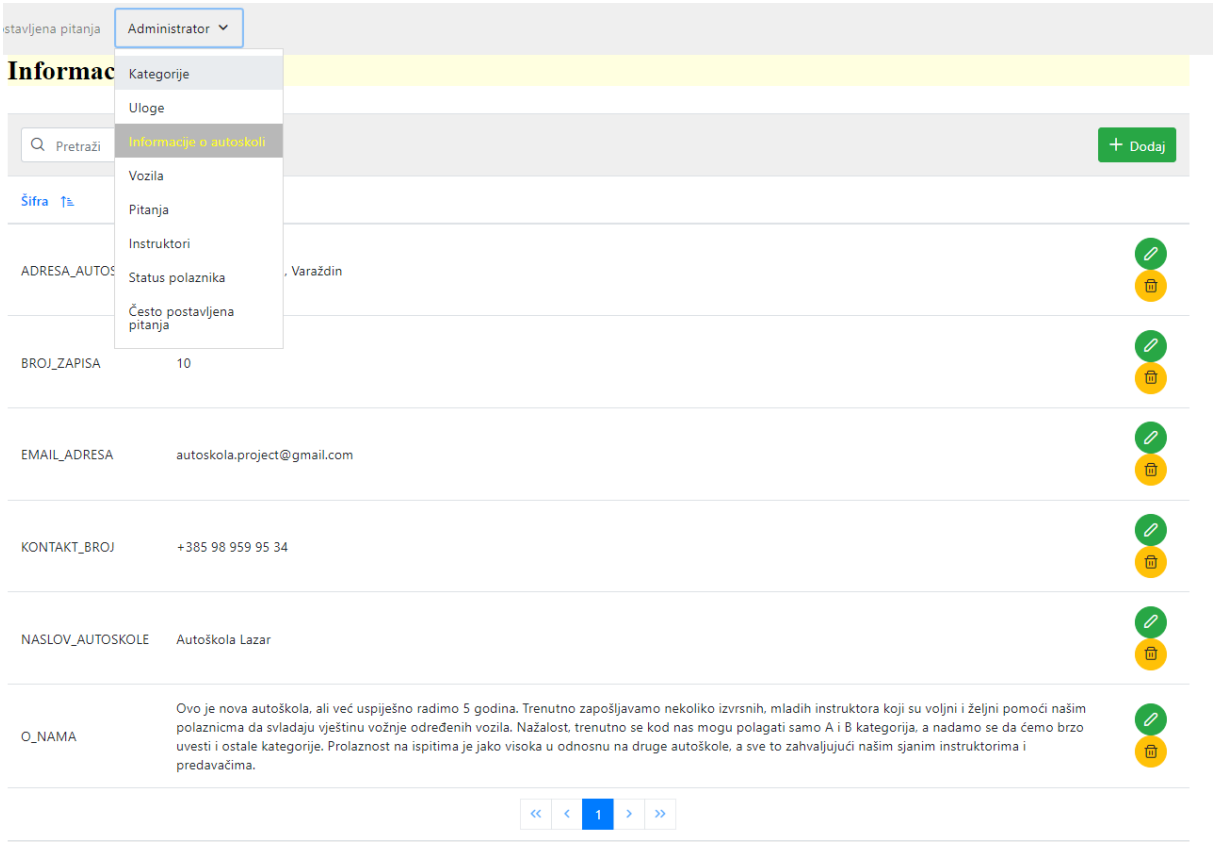
Obrazac za registraciju prikazana je na slici ispod. Za registraciju potrebno je unijeti sljedeće podatke: ime, prezime, e-mail adresu, korisničko ime i lozinku. Ako se unese korisničko ime ili e-mail koji već postoji, korisnik dobije poruku o neuspješnoj registraciji i razlog zbog kojeg je došlo do neuspješne registracije. Ukoliko su svi podaci u redu, registracija je uspješna i korisnik se preusmjeri na *Početnu stranicu*.



Slika 38. Obrazac za registraciju [Autorski rad]

Administratorove stranice

Administratorove stranice su stranice slične stranici za vođenje dnevnika. Dakle sve stranice su tablični prikazi entiteta iz tablica kojima administrator može upravljati. Kao što se može primijetiti na slici 39. koja prikazuje tablicu s osnovnim informacijama o autoškoli, kao što su: adresa, broj zapisa, e-mail adresa, kontakt broj, naslova autoškole i o nama. Te informacije administrator može dodavati, ažurirati i brisati, a implementacija je gotovo jednaka kao i kod vođenja dnevnika voženje. Osim te stranice administrator može još mijenjati kategorije, uloge, vozila, pitanja, instruktore, statuse polaznika te često postavljena pitanja.



Slika 39. Prikaz stranice s informacijama o autoškoli i ostalih administratorovih stranica [Autorski rad]

9.4. Kontejnerizacija aplikacije

Sad kad je aplikacija razvijena sa svim svojim funkcionalnostima, potrebno je kontejnerizirati svaki njezin dio. Kao što se spomenulo u opisu aplikacije, aplikacija se sastoji od tri dijela: korisnički dio, poslužiteljski dio te baza podataka. U ovom poglavlju će se prikazati kako se pomoću Dockera kontejnerizira svaki dio aplikacije. Za sljedeće akcije potrebno je imati instaliran Docker, kubectl command line tool i imati uključen Kubernetes na Dockeru.

9.4.1. Kontejnerizacija baze podataka

Aplikacija koristi MySQL bazu podataka te MySQL baza podataka već ima svoju kontejneriziranu verziju dostupno na Dockeru te je potrebno preuzeti njezinu sliku. Slika se preuzima na način da se otvori CMD (*eng. Command Promt*), ulogira u Docker te izvrši sljedeća naredba.

```
docker pull mysql:5.7
```

Kada se izvrši naredba iznad, Docker preuzima sliku *mysql* verzije 5.7 iz online Docker repozitorija, te se na Docker profilu sada nalazi MySQL kontejner.

9.4.2. Kontejnerizacija poslužiteljskog dijela aplikacije

Za kontejnerizaciju poslužiteljskog dijela potrebno je stvoriti tako zvani Dockerfile, odnosno Docker datoteku. No, prije nego se stvori Docker datoteka potrebno je izvršiti *očisti i izgradi* (*eng. Clean & Build*) nad poslužiteljskom objektu, tako da se dobije .jar izvršna datoteka. Primjer Docker datoteke za kontejnerizaciju prikazan je na slici ispod.

Ove naredbe objašnjene su u poglavlju Docker-tehnologija kontejnera. Dakle ovdje se preuzima osnovna slika Jave verzije 11, izlažu se vrata na vrijednosti 8080, dodaju se izvršne datoteke aplikacije autoškole te se definiraju naredbe i argumenti za izvršni kontejner. Nakon što je Docker datoteka definirana, potrebno je otvoriti CMD, premjestiti se u direktorij u kojem se nalazi ova Docker datoteka, te izvršiti sljedeću naredbu:

```
docker build -t autoskola-central-back:1.0 .
```

Riječ *docker* označava da se radi o Docker program. Riječ *build* je Dockerova naredba za izgradnju slike iz Docker datoteke. Dio naredbe *-t autoskola-central-back:1.0* označava pod kojom oznakom (*eng. Tag*) će se spremiti slika u Docker repozitoriju. U ovom slučaju sprema se pod nazivom *autoskola-central-back* i verzija slike je *1.0*, te cijela oznaka glasi *autoskola-central-back:1.0*. Točka (.) na kraju označava da se izgradnja slika događa u trenutačnom direktoriju.

*Dockerfile

```
1 FROM openjdk:11
2
3 EXPOSE 8080
4
5 ADD target/autoskola-central-back.jar autoskola-central-back.jar
6
7 ENTRYPOINT ["java","-jar","autoskola-central-back.jar"]
```

Slika 40. Docker datoteka za kontejnerizaciju poslužiteljskog dijela [Autorski rad]

9.4.3. Kontejnerizacija korisničkog dijela aplikacije

Kontejnerizacija korisničkog dijela aplikacije je vrlo slična kontejnerizaciji poslužiteljskog dijela aplikacije, jedino je razlika u Docker datoteci. Dakle, prvo je potrebno napraviti izgradnju aplikacije korisničkog dijela. To se može napraviti preko CMD-a, tako da se premjesti u direktorij gdje se nalazi aplikacija i pokrene sljedeća naredba:

```
npm run build
```

To je naredba *Node package manager-a* za izgradnju aplikaciji. Nakon toga je potreban Docker datoteka koja se nalazi na slici ispod. Ova datoteka ima samo dvije naredbe. Prva preuzima osnovnu *nginx* sliku *alpine* verzije, čija slika služi kao HTTP poslužitelj. Nakon toga se kopira izgrađeni dio aplikaciju u tu sliku. Naredba koja pokreće izgradnju je:

```
docker build -t autoskola-central-back:1.0 .
```

Naredba je jednaka kao i kod poslužiteljskog dijela, samo je razlika u oznaci slike, tj. nije *autoksola-central-back*, već je *autoskola-central-front*.

Dockerfile ×

Dockerfile

```
1   FROM nginx:alpine
2   COPY /dist/autoskola-central-front /usr/share/nginx/html
```

Slika 41. Docker datoteka za kontejnerizaciju korisničkog dijela [Autorski rad]

9.5. Orkestracija kontejnerima aplikacije

U ovom poglavlju će se pomoću Kubernetesa povezati aplikacija te tako biti dostupna za korištenje. Isto kao u prošlom poglavlju krenut će se s postavljanjem baze podataka pa zatim s poslužiteljskim dijelom te na kraju s korisničkim dijelom. U ovom poglavlju će se koristiti .yml datoteke i kubectl command line tool.

9.5.1. Upravljanje kontejnerom baze podataka

Baza podataka uvijek sadrži neke važne informacije koje ne bi trebao moći svatko vidjeti. Iz tog razloga da bi se upravljalo bazom podataka treba se moći prijaviti u bazu. Kubernetes podatke o prijavi čuva u Tajnama (*eng. Secrets*). Kubernetes Tajna je objekt koji sadrži malu količinu osjetljivih podataka kao što su lozinka, token ili ključ. Da nema Tajni, osjetljivi podaci bi trebali biti smješteni u programski kôd i tako bi bili nezaštićeni. Tajne se kreiraju neovisno o Podovima pa se tako smanjuje rizik o otkrivanju osjetljivih podataka. Kod aplikacije Autoškola, kreirat će se dvije Tajne, jedna s podacima o administratoru baze podataka i jedna o korisniku baze podataka.

Ispod se nalaze dvije tajne: *mysqldb-root-credentials.yml* i *mysql-credentials.yml*. U prvoj se nalazi lozinka za korijenskog korisnika, koja je hashirana pomoću Base64 vrijednosti, a u drugoj se nalaze podaci o korisniku koji pristupa bazi podataka. Njegovo korisničko ime i lozinka, također su hashirani pomoću Base64 vrijednosti.

Svaka yml datoteka ima vrijednosti *apiVersion*, *kind* i *metadata*. Vrijednost *apiVersion* označava verziju, *kind* označava vrstu, a *metadata* metapodatke datoteke. U ovom slučaju vrijednost od *kind* je *Secret* što označava Kuberentesu da se radi o Tajni.

mysqldb-root-credentials.yml

```
apiVersion: v1
kind: Secret
metadata:
      name: db-root-credentials
data:
     password: cm9vdA==
```

mysqldb-credentials.yml

```
apiVersion: v1
kind: Secret
metadata:
      name: db-credentials
data:
      username: cm9vdA==
       password: cm9vdA==
```

Sljedeća datoteka koja će se definirati je konfiguracijska mapa (*eng. ConfigMap*) te se ispod nalazi sadržaj konfiguracijske mape. Kubernetesova konfiguracijska mapa je objekt u kojeg se spremaju neosjetljivi podaci u paru ključ-vrijednost. U ovoj konfiguracijski mapi spremljeni su podaci o nazivu baze podataka i nazivu sheme. Može se primijetiti da za razliku od prijašnjih datoteka, vrsta ove datoteke nije *Secret*, već *ConfigMap* te Kubernetes drugačije tretira ovu datoteku.

mysqldb-configmap.yml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-conf
data:
 host: mysql
 name: autoskola
```

Sljedeća datoteka se sastoji od tri vrste Kubernetesovih objekta: servisa, *PresistentVolumeClaim* i *Deployment*. Datoteka se sastoji od svih tri objekta, ali ovdje će se objekti razdvojiti kako bi se detaljnije svaki od njih mogao pojasniti. Prvo će se definirati servis.

Kubernetes servis pruža apstraktan način da izlaganja aplikacije koja se pokreće na skupini Podova. Vrsta ovog objekta je servis, te važni podaci koji su definirani u ovaj datoteci su naziv metapodatka, jer on definira DNS naziv servisa. Potrebno je definirati na kojim vratima se nalazi servis kako bismo mu mogli poslije pristupiti. Te klaster IP je postavljen na *None* zbog toga što će se bazi podataka pristupati preko DNS naziva.

mysqldb-deployment.yml

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    app: mysql
    tier: database
spec:
  ports:
    - port: 3306
      targetPort: 3306
  selector:
    app: mysql
    tier: database
  clusterIP: None
```

Sljedeće slijedi *PresistentVolumeClaim* objekt. *PresistentVolumeClaim* definira zahtjev korisnika za spremište. U ovom objektu korisnik definira kojim načinom rada će spremište raditi i kolika će spremište biti veliko. Načini rada mogu biti *ReadWriteOnce, ReadOnlyMany, ReadWriteMany*, itd. U primjeru aplikacija Autoškola, u *PresistentVolumeClaim* objektu definirao se način rada *ReadWriteOnce*, što označava da se može pristupiti bazi podataka samo s isto čvora, i da memorija spremišta bude 1 gibibajt.

mysqldb-deployment.yml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
  labels:
    app: mysql
    tier: database
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

I za kraj će se definirati *Deployment*. *Deployment* pruža deklarativna ažuriranja za Podove i setove replika. U *Deploymentu* se opisuje željeno stanje, a kontroler se brine o tome da mijenja stvarno stanje u željeno stanje. Također se može definirati koliko Podova i replika će biti u implementaciji. Ovdje treba pripaziti na nekoliko stvari. Dio koji je označen sa (1) u datoteci treba sadržavati jednake oznake kao mysql Pod. Dio označen sa (2) označava da servis i *Deployment* moraju imati jednake oznake. (3) Označava sliku koja se preuzima iz Dockera. (4) Preuzima lozinku korijenskog korisnika iz Tajne. (5) Naziv Tajne. (6) Ključ u

Tajni koji sadrži zadanu vrijednost. (7) Postavljanje naziv baze podataka iz konfiguracijske mape. (8) Postavljanje *PresitentVolumeClaima*. (9) Prosljeđivanje „*Volume-a*“ iz PVC objekta.

mysqldb-deployment.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  labels:
    app: mysql
    tier: database
spec:
  selector: #(1)
    matchLabels:
      app: mysql
      tier: database
  strategy:
    type: Recreate
  template:
    metadata:
      labels: #(2)
        app: mysql
        tier: database
    spec:
      containers:
      - image: dnhsoft/mysql-utf8:5.7 #(3)
        args:
          - "--ignore-db-dir=lost+found"
        name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD #(4)
          valueFrom:
            secretKeyRef:
              name: db-root-credentials #(5)
              key: password   #(6)
        - name: MYSQL_DATABASE #(7)
          valueFrom:
            configMapKeyRef:
              name: db-conf
              key: name
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:  #(8)
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage #(9)
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

Kad su se definirale sve potrebne datoteke potrebno ih je primijeniti u Kuberentesu. Za to je potrebno u CMD-u postaviti se u direktorij gdje se nalaze te .yml datoteke te izvršiti sljedeće naredbe:

```
kubectl apply -f mysqldb-root-credentials.yml
kubectl apply -f mysqldb -credentials.yml
kubectl apply -f mysqldb-configmap.yml
kubectl apply -f mysqldb-deployment.yml
```

Nakon što su se sve ove naredbe izvršile, u Kubernetes klasteru su se pojavile nove Tajne, Podovi, setovi replika, konfiguracijska mapa te *Deployment*.

9.5.2. Upravljanje kontejnerom poslužiteljskog dijela

Kod upravljanja kontejnera poslužiteljskog dijela koristi će se samo jedna datoteka koja sadrži dva objekta, servis i *Deployment*. Oni su pokazani ispod u datoteci dpl.yml. Ovdje za razliku od servisa baze podataka osim što se definirala vrata u samom klasteru, ovdje se definiraju vrata čvora. Vrata čvora izlažu servis na svakom čvoru na jednakim vratima, odnosno servis je dostupan na <IP-čvora>:<vrata-čvora> adresi.

dpl.yml

```yaml
kind: Service
apiVersion: v1
metadata:
  name: autoskola-central-back
  labels:
    name: autoskola-central-back
spec:
  ports:
    - nodePort: 30163
      port: 8080
      targetPort: 8080
      protocol: TCP
  selector:
    app: autoskola-central-back
  type: NodePort
```

Kod *Deploymenta* se definira sljedeće : (1) Broj replika. (2) Naziv slike u Dockeru. (3) Postavljanje varijabli okruženja. (4) Postavljanje adrese baze podataka iz konfiguracijske mape. (5) Naziv konfiguracijske mape. (6) Postavljanje naziva baze podatka iz

konfiguracijske mape. (7) Postavljanje korisničkog imena iz Tajne. (8) Naziv Tajne. (9) Postavljanje lozinke baze podataka iz Tajne.

dpl.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: autoskola-central-back
spec:
  selector:
    matchLabels:
      app: autoskola-central-back
  replicas: 1  # (1)
  template:
    metadata:
      labels:
        app: autoskola-central-back
    spec:
      containers:
        - name: autoskola-central-back
          image: autoskola-central-back:1.0 # (2)
          ports:
            - containerPort: 8080
          env:   # (3)
          - name: DB_HOST   # (4)
            valueFrom:
              configMapKeyRef:
                name: db-conf  #(5) name of configMap
                key: host
          - name: DB_NAME  #(6)
            valueFrom:
              configMapKeyRef:
                name: db-conf
                key: name
          - name: DB_USERNAME  #(7)
            valueFrom:
              secretKeyRef:
                name: db-credentials #(8)
                key: username
          - name: DB_PASSWORD #(9)
            valueFrom:
              secretKeyRef:
                name: db-credentials
                key: password
```

Nakon što postavi ova .yml datoteka, potrebno je primijeniti tu datoteku u Kubernetesu kako bi dobili servis i *Deployment* za poslužiteljski dio. Dakle, u CMD-u se postavi u direktoriju gdje je ta datoteka i izvrši se naredba:

```
kubectl apply -f dpl.yml
```

9.5.3. Upravljanje kontejnerom korisničkog dijela

Ovdje će biti prikazana i ukratko objašnjena .yml datoteka za korisnički. Jako je slična datoteci poslužiteljskog dijela. Ispod se nalazi *autoskola-central-front.yml* datoteka u kojoj se nalaze *Deployment* i servis. (1) Kubernetes API verzija. (2) Vrsta Kubernetes objekta. (3) Broj replika koje će biti kreirane. (4) Docker slika koje će se koristiti. (5) Vrata na kojim će se pokretati kontejner na klasteru. (6) Vrata na kojem će se pokretati servis u klasteru. (7) Vrata koje izlaže servis. (8) Vrsta servisa.

autoskola-central-front.yml

```
apiVersion: apps/v1 # (1)
kind: Deployment  # (2)
metadata:
  name: autoskola-central-front
spec:
  selector:
    matchLabels:
      app: angular
  replicas: 3 # (3)
  template:
    metadata:
      labels:
        app: angular
    spec:
      containers:
      - name: angular
        image: autoskola-central-front:1.0 #(4)
        ports:
        - containerPort: 80 # (5)
---

kind: Service  # (2)
apiVersion: v1 # (1)
metadata:
  name: autoskola-central-front
spec:
  selector:
    app: angular
  ports:
  - protocol: TCP
    port: 80   # (6)
    targetPort: 80   # (7)
    nodePort: 31000
  type: NodePort     #(8)
```

Naredba kojom se pokreće :

```
kubectl apply -f autoskola-central-front.yml
```

10. Zaključak

Danas se više ne može reći da je Web popularan, jer Web je postao dio svakodnevice. Gotovo svako poduzeće danas ima svoju Web stranicu, odnosno Web aplikaciju, ovisno o potrebi poduzeća. Vrijeme pandemije prisililo je ljude da više-manje sve rade iz svoje kuće što je povećalo posjet Web stranicama. S toga potreba za Web aplikacijama je jako velika. Svaki klijent koji želi imati Web aplikaciju, želi da njegov proizvod bude što prije isporučen, što kvalitetniji i da košta što manje. Sve želje klijenta je nemoguće ispuniti pa se klijenti često odluče na brzinu isporuke, a zbog toga pada kvaliteta proizvoda. Zato je važno imati što bolje programske okvire kako bi se mogle Web aplikacije brže, bolje i kvalitetnije napraviti i isporučiti.

Spring programski okvir je okvir otvorenog kôda te se razvilo dosta novih programskih okvira koje koriste značajke Springa s nekom svojom nadogradnjom. Usprkos tome Spring je dalje među najpopularnijima okvirima za izradu Web aplikacije, a tome mu pridonosi velika zajednica, puno dokumentacije i edukativnih videa koje razvojni programeri mogu koristiti. Popularnosti Spring pridonosi i brzina podizanja poslužiteljske aplikacije, jer sam Spring se sa svojim značajkama pobrinuo da programeri što manje brinu o konfiguracijama pa se mogu više fokusirati na poslovnu logiku.

Slično slučaj je i s Dockerom i Kubernetesom. Osim same izrade aplikacije, klijentu je bitna i isporuka, a isporuka se danas najbolje i najbrže pruža kontejnerizacijom. Docker i Kuberentes su također aplikacije otvorenog kôda te su se razvile slične aplikacije, ali dalje Docker i Kubernetes su najpopularniji izbor kod mnogih programera. Također imaju veliku zajednicu, puno dokumentacije i edukativnih videa. Ali nije riječ samo o tome, nego i o njihovoj brzini i efikasnosti. Jer Docker, Kubernetes i Spring se i danas razvijaju, dobivaju nove značajke kako bi i dalje ostale konkurentne na tržištu.

Popis literature

[1] T. Berners-Lee, „Information Management: A Proposal“, *CERN*, izd. March 1989, str. 20.

[2] A. Grosskurth i M. W. Godfrey, „Architecture and evolution of the modern web browser“. Elsevier Science: School of Computer Science, University of Waterloo, str. 24.

[3] K. C. A. Khanzode, „EVOLUTION OF THE WORLD WIDE WEB: FROM WEB 1.0 TO 6.0“, International Journal of Digital Library Services, 2250-1142 sv. 6, str. 11, 2016.

[4] D. Liu, S. Ma, Q. Ru, i Z. Guo, „Design and realization of network education supporting system based on web 2.0“, u *2009 2nd IEEE International Conference on Computer Science and Information Technology*, kol. 2009, str. 374–377. doi: 10.1109/ICCSIT.2009.5234925.

[5] M. Grgić (2021), „ARHITEKTURA MODERNIH WEB APLIKACIJA NA ASP.NET CORE PLATFORMI“ (Diplomski rad), Fakultet elektrotehnike, računarstva i informacijskih tehnologija, Sveučilište Josipa Jurja Strossmayera u Osijeku

[6] V. Solovei, O. Olshevska, i Y. Bortsova, „THE DIFFERENCE BETWEEN DEVELOPING SINGLE PAGE APPLICATION AND TRADITIONAL WEB APPLICATION BASED ON MECHATRONICS ROBOT LABORATORY ONAFT APPLICATION“, *ATBP*, sv. 10, izd. 1, tra. 2018, doi: 10.15673/atbp.v10i1.874.

[7] J. Yang, „Web Service Componentization“. Communications of the ACM , str. 35–40, lis. 2003, doi: 10.1145/944217.944235.

[8] D. Booth, H. Hass, i F. McCabe (2004), „Web Services Architecture“. Preuzeto s: https://www.w3.org/TR/ws-arch/#whatis (pristupljeno 08. srpanj 2022.).

[9] P. Marić (2018), „PROGRAMSKOG SUČELJA U .NET OKVIRU“ (Završni rad), Fakultet elektrotehnike, računarstva i informacijskih tehnologija, Sveučilište Josipa Jurja Strossmayera u Osijeku

[10] B. Balentović (2018), „SOAP protokol i njegova primjena“(Završni rad), Fakultet elektrotehnike, računarstva i informacijskih tehnologija, Sveučilište Josipa Jurja Strossmayera u Osijeku

[11] A. Walker (2020), „RESTful Web Services Tutorial: What is REST API with Example“, Preuzto s:. https://www.guru99.com/restful-web-services.html (pristupljeno 14. srpanj 2022.).

[12] C. Pautasso, „RESTful Web services: principles, patterns, emerging technologies“, Faculty of Informatics, University of Lugano

[13] K. I. Fagerl, F. Mantz, R. Rossini, i A. Rutle (2010), „Groovy and Grails Meet Eclipse Modelling Framework“, NIK-2010 conference, Bergen, Norway

[14] M. Grönroos (2014), *Book of Vaadin: Vaadin 7 Edition - 6th Revision*, Vaadin Ltd (7. izd.) 2015.

[15] D. Mane, K. Chitnis, i N. Ojha (2013), „The Spring Framework: An Open Source Java Platform for Developing Robust Java Applications“. International Journal of Innovative Technology and Exploring Engineering sv. 3, izd. 2, 2278-3075

[16] R. Johnson *i ostali (2013)*, „Spring Framework Reference Documentation“ (Verzija 3.2.17) Preuzeto s: https://docs.spring.io/spring-framework/docs/3.2.17.RELEASE/spring-framework-reference/htmlsingle/

[17] K. S. P. Reddy (2017), *Beginning Spring Boot 2: Applications and Microservices with the Spring Framework*. Hyderbad,India: Apress

[18] P. Srivastava (2020), „Difference between Spring and Spring Boot“, *GeeksforGeeks*, Prezeto s: https://www.geeksforgeeks.org/difference-between-spring-and-spring-boot/ (pristupljeno 05. kolovoz 2022.).

[19] D. Zhang, Z. Wei, i Y. Yang (2013), „Research on Lightweight MVC Framework Based on Spring MVC and Mybatis“, u *2013 Sixth International Symposium on Computational Intelligence and Design*, lis. 2013, sv. 1, str. 350–353. doi: 10.1109/ISCID.2013.94.

[20] A. Poutsma, R. Evans, i T. Abed Rabbo (2014), „Spring Web Services Reference Documentation“. Preuzeto s: https://docs.spring.io/spring-ws/docs/2.4.0.RELEASE/reference/htmlsingle/#what-is-spring-ws (pristupljeno 05. kolovoz 2022.).
[21] J. Turnbull (2014), *The Docker Book*. 2014. (Verzija 1.2.0), preuzeto s: https://lsi.vc.ehu.eus/pablogn/docencia/manuales/The%20Docker%20Book.pdf
[22] M. Aleksic (2022), „Podman vs Docker: Everything You Need to Know | phoenixNAP KB“, *Knowledge Base by phoenixNAP*, preuzeto s: https://phoenixnap.com/kb/podman-vs-docker (pristupljeno 25. kolovoz 2022.).
[23] N. Poulton (2020), „The Kubernetes Book“, Leanpub str. 182.
[24] H. A. Ozmen (2018), „Design and implementation of an Iot-based home automation system utilizing fog and cloud computing paradigms“ (Diplomski rad), B.S., Computer Engineering, Bo˘gazi¸ci University

Popis slika

Popis tablica